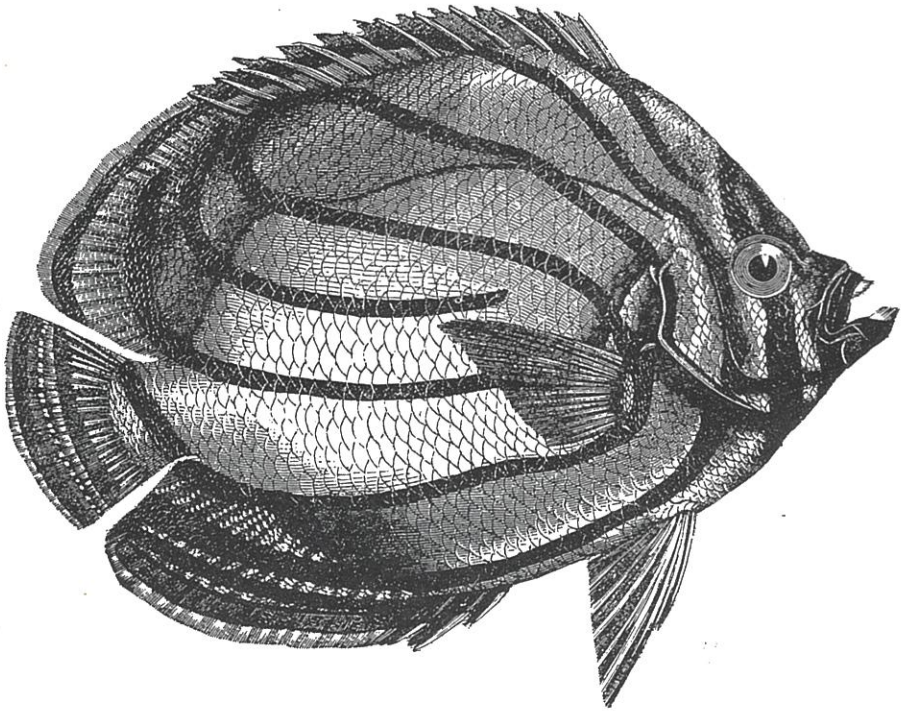
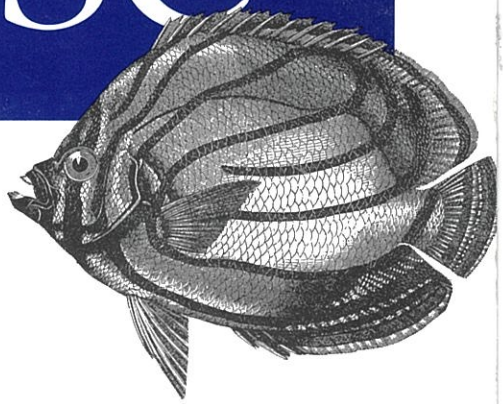
Développement d'applications Java

Édition française

Eclipse





Steve Holzner

Traduction de Denis Priou & Joël Senpauroca

O'REILLY®

Eclipse

Eclipse

Steve Holzner

O'REILLY®

Cambridge • Cologne • Farnham • Paris • Pékin • Sebastopol • Taipei • Tokyo

Eclipse
Steve Holzner

L'édition originale de ce livre (ISBN 0-596-00641-1) a été publié aux États-Unis sous le titre : *Eclipse*

© O'REILLY MEDIA Inc., 2004

Couverture conçue par Ellie VOLCKHAUSEN et Hanna DYER

Édition : Florence LEROY

Traduction : Denis PRIOU et Joël SENPAUROCA

HISTORIQUE :

Décembre 2004 : 1re édition

Les programmes figurant dans ce livre ont pour but d'illustrer les sujets traités. Il n'est donné aucune garantie quant à leur fonctionnement une fois compilés, assemblés ou interprétés dans le cadre d'une utilisation professionnelle ou commerciale.

Toute représentation ou reproduction, intégrale ou partielle, faite sans le consentement de l'auteur, de ses ayants droit, ou ayants cause, est illicite (loi du 11 mars 1957, alinéa 1er de l'article 40). Cette représentation ou reproduction, par quelque procédé que ce soit, constituerait une contrefaçon sanctionnée par les articles 425 et suivants du Code pénal. La loi du 11 mars 1957 autorise uniquement, aux termes des alinéas 2 et 3 de l'article 41, les copies ou reproductions strictement réservées à l'usage privé du copiste et non destinées à une utilisation collective d'une part et, d'autre part, les analyses et les courtes citations dans un but d'exemple et d'illustration.

© ÉDITIONS O'REILLY, Paris, 2004.
ISBN 2-84177-264-0

Table des matières

Préface ix

1. *Eclipse en quelques mots* 1
   - Eclipse et Java 1
   - Installation d'Eclipse 4
   - Qu'est-ce qu'Eclipse ? 6
   - Vues et perspectives 9
   - Travailler sous Eclipse 11
   - Utilisation du correctif rapide (fonction Quick Fix) 22
   - Deux mots sur la gestion de projets 25

2. *Développement d'applications Java* 29
   - Développement de code Java 29
   - Compilation et exécution du code 39
   - Création d'un document Javadoc 48
   - Propagation des modifications 48
   - Fonctionnalités complémentaires 55
   - Personnalisation de l'environnement de développement 58

3. *Test de bon fonctionnement d'une application et débogage* 65
   - Test de bon fonctionnement d'applications Java : l'outil JUnit 65
   - Débogage 75

4. *Développement d'applications en équipe sous Eclipse* 93
   - Principes du contrôle du code source 93

Les principes de fonctionnement de CVS 93
Recherche d'un serveur CVS 95
Ajout d'un projet dans le référentiel CVS 96

5. *Compilation d'un projet Eclipse au moyen de Ant* 115
Travailler avec Ant 115
Production de fichiers JAR 118
Configuration de Ant dans Eclipse 124
Détection des erreurs dans les fichiers de compilation 127

6. *Programmation GUI : des applets à Swing* 131
Création d'une application AWT 134
Création d'applications Swing 136
Utilisation de modules Eclipse 142
Utilisation du module V4ALL 143

7. *SWT : boutons, textes, labels, listes, gestionnaires de placement et evénements* 149
Graphiques Java 149
Un exemple SWT 150
Utilisation des boutons 158
Utilisation des éléments composites et des gestionnaires de placement 163
Utilisation des Listes 165
Utilisation de V4ALL avec SWT 167

8. *SWT : menus, barres d'outils, curseurs, arbres et boîtes de dialogue* 171
Utilisation des menus 171
Utilisation de la barre d'outils 176
Utilisation de curseurs 180
Utilisation des arbres 184
Utilisation des boîtes de dialogue 187
Ouverture d'Internet Explorer dans une fenêtre SWT 192

9. *Développement d'applications pour le Web* 197
Installation et configuration de Tomcat 197
Création d'une JSP 199

Création d'un servlet 201
Création d'un servlet avec gestion automatique des répertoires 204
Connexion à un JavaBean 207
Utilisation du plug-in Tomcat Sysdeo 209
Déploiement d'applications pour le web 216

10. *Développement d'applications Struts avec Eclipse* 221
Struts et Eclipse 222
Création de la Vue 225
Création du Contrôleur 228
Création du Modèle 231
Utilisation du plug-in Easy Struts 236

11. *Réalisation d'un plug-in : environnement de développement de plug-in, fichiers manifestes et points d'extension* 245
Le fichier manifeste plugin.xml 246
Utilisation de l'environnement de développement de plug-in 247
Utilisation du plan de travail d'exécution 251
Création d'un plug-in standard 253

12. *Développement d'un plug-in: Création d'éditeurs et de vues* 265
Création d'un éditeur multipage 265
Création d'une vue 273
Déploiement d'un plug-in 280

13. *Eclipse 3.0* 283
Un survol d'Eclipse 3.0 283
Création d'un projet Java 284
Modifications dans la plateforme Eclipse 288
Modifications effectuées dans les outils de développement Java 292
Autres modifications 299

Index 301

Préface

Bienvenue dans le monde d'Eclipse, le plus populaire des environnements de développement intégrés (EDI) dédiés au langage Java. Éclipse est un outil extraordinaire qui vient satisfaire les besoins exprimés de longue date par les développeurs Java. Grâce à lui, c'en est terminé des longues pages d'erreurs défilant hors des limites de l'écran, suite au lancement du compilateur en ligne de commande. Avec Eclipse, le programmeur dispose d'un EDI qui se charge de tout, le programmeur pouvant alors se concentrer exclusivement sur l'écriture de son code. Ceux qui n'ont jamais utilisé Eclipse auparavant verront leur productivité bondir d'un pas de géant.

L'objectif de ce livre est de présenter Eclipse en détail, des traitements de base jusqu'aux techniques les plus avancées. Cet ouvrage se veut pédagogique avant tout. Rédigé par un programmeur, il est destiné à des programmeurs et a pour but d'accélérer leur apprentissage de l'outil Eclipse.

Cet ouvrage est rédigé afin de fournir aux programmeurs les informations qu'ils recherchent : le meilleur, et seulement lui. Les éléments décrits dans ce livre leur permettront de maîtriser parfaitement les nombreuses possibilités offertes par Eclipse, ce qui est notre objectif.

Contenu de l'ouvrage

Cet ouvrage présente des centaines de techniques entièrement destinées à mieux utiliser Eclipse, en partant des techniques élémentaires du développement sous Java et en allant jusqu'à la création des éditeurs plug-in dédiés à l'environnement Eclipse. Voici quelques-uns des thèmes qui seront abordés dans cet ouvrage :

- Utilisation d'Eclipse pour développer du code Java
- Manipulation de fichiers JAR
- Constitution de configurations de lancement
- Sélection de la machine virtuelle Java
- Création de fichiers Javadoc
- Propagation des modifications
- Extraction d'interfaces
- Visualisation de hiérarchies de types

- Personnalisation d'Eclipse
- Validation de code au moyen de JUnit
- Débogage
- Création de compteurs sur les points d'arrêt
- Édition de code en cours de débogage
- Partage de projets avec CVS
- Comparaison de code avec l'historique local
- Utilisation de Ant pour construire des projets Eclipse
- Programmation d'interfaces graphique utilisateur (GUI), des applets à Swing
- Utilisation du Standard Widget Toolkit (SWT)
- Boutons, textes, étiquettes, listes, gestionnaires de placement et événements SWT
- Menus, barres à outils, glissières, arborescences et boîtes de dialogue SWT
- Développement d'applications Web
- Écriture de code servlet
- Utilisation du plug-in Sysdeo Tomcat
- Débogage et déploiement de projets Web
- Développement d'applications Struts avec Eclipse
- Utilisation du plug-in Easy Struts
- Développement de plug-in Eclipse
- Le Plug-in Development Environment (PDE)
- Les fichiers manifestes de plug-in
- Les points d'extension
- Utilisation du Run-time Workbench
- Création d'un plug-in standard
- Création d'un actionSet
- Création de menus plug-in
- Création d'un plug-in éditeur multi-fenêtre
- Création d'un assistant de plug-in
- Création d'une vue Eclipse supportée par un plug-in

On abordera tous ces sujets, et bien d'autres encore, dans les pages suivantes. Voici une vue d'ensemble des chapitres de cet ouvrage :

Chapitre 1, *Eclipse en quelques mots*

Ce chapitre couvre les bases de l'EDI Eclipse ; il détaille l'espace de travail d'Eclipse et constitue une introduction au JDT (*Java Development Tools* – outils de développement Java).

Chapitre 2, *Développement d'applications Java*

Ce chapitre décrit la création de projets Java, la création automatique de code, l'outil de vérification de syntaxe et le développement de petites applications Java au moyen du JDT.

Chapitre 3, *Test de bon fonctionnement d'une application et débogage*

Un EDI ne serait rien sans un débogueur. Eclipse et le JDT fournissent un débogueur puissant, incluant les points d'arrêt, l'évaluation d'expressions, la modification de valeurs en cours d'exécution et bien plus encore.

Chapitre 4, *Développement d'applications en équipe sous Eclipse*

L'un des grands avantages d'Eclipse est de faciliter le développement en équipe, chose très appréciable pour tout développeur professionnel, puisque tout développement informatique d'importance est généralement réalisé en équipe. En utilisant l'outil CVS (Concurrent Versions System), les membres d'une équipe peuvent travailler sur leur propre poste tout en partageant et en archivant leur travail au moyen du référentiel CVS.

Chapitre 5, *Compilation d'un projet Eclipse au moyen de Ant*

Ant est un puissant outil de construction d'applications Java. Ce chapitre montre ce qu'il est possible de faire en utilisant conjointement Ant et Eclipse, comme la copie et le déplacement de fichiers, et qu'Eclipse seul ne peut réaliser.

Chapitre 6, *Programmation GUI : des applets à Swing*

Ce chapitre décrit les techniques de développement de GUI avec Swing en utilisant Eclipse. On examinera aussi un plug-in Swing pour Eclipse – l'utilisation de ce plug-in permet, par exemple, de déplacer des contrôles Swing par glisser-déposer (*drag and drop*).

Chapitre 7, *SWT : boutons, textes, labels, listes, gestionnaires de placement et évenements*

Ce chapitre (ainsi que le suivant) couvre le SWT (*Standard Widget Toolkit*). Cette boîte à outils dédiée aux GUI et destinée aux développeurs Java fournit une API portable et assure une intégration solide dans la plate-forme GUI native sous-jacente. SWT constitue une alternative à AWT et Swing – c'est pour cela, entre autres, qu'Eclipse est devenu si populaire ; deux chapitres seront consacrés à ce sujet.

Chapitre 8, *SWT : menus, barres d'outils, curseurs, arbres et boîtes de dialogue*

Ce chapitre complète la description de SWT. Y sont abordés les boîtes de dialogue, les barres à outils, les glissières, les arborescences, les menus, etc.

Chapitre 9, *Développement d'applications pour le Web*

Ce chapitre aborde le développement d'applications pour le Web, ainsi que la création de servlets et de JSP au moyen d'Eclipse et du serveur Tomcat. On examinera aussi un plug-in Tomcat Eclipse.

Chapitre 10, *Développement d'applications Struts avec Eclipse*

Le développement d'applications pour le Web se poursuit dans ce chapitre, dans lequel on présente l'ossature Struts. On verra aussi comment utiliser un plug-in Struts.

Chapitre 11, *Réalisation d'un plug-in : environnement de développement de plug-in, fichiers manifestes et points d'extension*

On aborde dans ce chapitre la création de plug-in Eclipse destinés à étendre les fonctionnalités d'Eclipse. À cette fin, on utilise le PDE (*Plug-in Developement Environment*) ainsi que d'autres outils de création de plug-in.

Chapitre 12, *Développement d'un plug-in : création d'éditeurs et de vues*

On aborde dans ce chapitre les détails de certains aspects parmi les plus avancés concernant la création de plug-in – éditeurs, création d'assistants, gestion de vues Eclipse.

Chapitre 13, *Eclipse 3.0*

Ce chapitre présente succinctement la version 3.0 d'Eclipse. Les perspectives d'évolution de ce logiciel y sont survolées.

Conventions utilisées dans cet ouvrage

Différentes conventions doivent être connues du lecteur, afin de faciliter la lecture de cet ouvrage. Lorsque des lignes de code sont ajoutées dans un listing et lorsqu'elles font l'objet d'une discussion, elles apparaissent en caractères gras, et lorsque du code additionnel reste à venir, l'espace dans lequel il va s'insérer est représenté par trois points de suspension. Par exemple :

```
Listener listener = new Listener( ) {
    public void handleEvent(Event event) {
        ToolItem item = (ToolItem)event.widget;
        String string = item.getText( );
            .
            .
            .
    }
};
```

Dans cet ouvrage, on utilise la convention standard de sélection d'un choix de menu ; par exemple, pour créer un nouveau projet sous Eclipse, on sélectionne le choix de menu Fichier→ Nouveau→ Projet.

Les conventions typographiques suivantes sont utilisées tout au long de cet ouvrage :

Le Monde Livre

Indique les titres de menus, les options de menus, les boutons de menus et les accélérateurs de clavier.

Le Monde Livre Italique

Indique les termes nouveaux, les URL ou les adresses électroniques données en exemple, les noms de fichiers, les extensions de fichiers, les chemins d'accès, les répertoires et les utilitaires Unix.

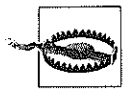
`The SansMono Condensed`

Indique les commandes, les options, les variables, les types, les classes, les espaces de nommage, les méthodes, les modules, les propriétés, les paramètres, les valeurs, les objets, les événements, les gestionnaires d'événements et les balises XML.

`The SanMono Condensed Italique`

Indique un texte qui doit être remplacé par des valeurs fournies par l'utilisateur.

Cette icône signale un conseil, une suggestion ou une remarque d'ordre général.



Cette icône signale un avertissement ou une mise en garde.

Logiciels requis

Tous les logiciels nécessaires à l'exploitation du code listé dans cet ouvrage peuvent être téléchargés gratuitement sur Internet. Le logiciel Eclipse est lui-même nécessaire – cet ouvrage a été écrit en utilisant la version 2.1.1 d'Eclipse – et le chapitre 1 décrit la marche à suivre pour le télécharger. D'autres logiciels s'avèreront utiles dans la suite du livre, en particulier le serveur Web Tomcat et divers serveurs CVS (qui permettent de partager des projets Eclipse).

Par construction, Eclipse est *extensible* ; des centaines de *plug-in* Eclipse sont d'ailleurs téléchargeables gratuitement. Ces plug-in permettent d'ajouter des fonctionnalités à Eclipse – de nouveaux éditeurs intégrés, des générateurs de code, des lanceurs logiciels, etc. – et un certain nombre d'entre eux, parmi les plus populaires, seront examinés dans cet ouvrage. Des plug-in Eclipse spécifiques seront par ailleurs développés dans le chapitre 11 et dans le chapitre 12.

Exemples

Tous les programmes listés dans cet ouvrage peuvent être téléchargés à partir de *http://www.oreilly.com/catalogue/2841772640.html/*. On consultera le fichier *lisezmoi.txt* pour obtenir toutes les informations relatives à l'installation de ces exemples.

Commentaires et questions

Nous avons apporté le maximum de soin à l'édition de cet ouvrage. Cependant, si vous rencontrez des erreurs, des inexactitudes ou si certains passages vous semblent confus, n'hésitez pas à nous les signaler à :

france@oreilly.com

Vous contribuerez ainsi à l'amélioration de la prochaine édition.

Une page web est consacrée à ce livre :

http://www.oreilly.com/catalogue/2841772640.html/

Pour être tenu au courant de nos publications

Site web

Pour de plus amples informations sur nos ouvrages, consultez notre site :

http://www.oreilly.fr/

Liste de diffusion par messagerie électronique

Si vous souhaitez recevoir périodiquement les annonces de nouveaux produits O'Reilly en français, il vous suffit de souscrire un abonnement à la liste d'annonces

`parutions-oreilly`, en envoyant un message électronique à *majordomo@ora.de* contenant (dans le corps du message) :

```
subscribe parutions-oreilly votre_adresse_email
```

Exemple :

```
subscribe parutions-oreilly jean.dupond@ici.fr
```

Cette liste ne véhicule que des annonces. Son volume ne dépassera pas quatre messages par mois.

CHAPITRE 1

Eclipse en quelques mots

Le lecteur de ce livre est très certainement un programmeur Java ; il sait combien ce langage peut être vétilleux. Des lignes d'`import` ou des déclarations de variables oubliées, des points-virgules omis, une syntaxe inexacte, des fautes de frappe : tous ces problèmes amènent le compilateur en ligne de commande, `javac`, à réagir violemment en affichant des pages d'erreurs rébarbatives. Ces messages indiquent au programmeur que `javac` a reconnu les erreurs, sans toutefois les corriger lui-même, empêchant ainsi le programmeur de se concentrer sur le vrai travail de développement.

En effet, `javac` n'est pas un éditeur : il ne peut pas résoudre le problème seul. Ces longs flots d'erreurs défilant à l'écran sont une expérience quotidienne pour le programmeur Java, qui peut accuser Java de manquer de souplesse. Pour en finir avec cela, il peut utiliser un environnement de développement intégré (EDI), qui non seulement identifiera les erreurs avant la compilation du code, mais qui formulera aussi des suggestions de solutions. Parmi les nombreux environnements de développement intégrés disponibles pour Java, Eclipse est à ce jour le meilleur. La figure 1-1 présente l'interface graphique utilisateur proposée par Eclipse.

Eclipse et Java

Eclipse se définit comme une *plateforme de développement universelle*, apte à supporter les EDI de toutes sortes de langages de programmation – des EDI sont disponibles pour des langages allant du C/C++ au Cobol –, mais c'est l'EDI Java qui est attaché à Eclipse, sujet principal de cet ouvrage, et qui concerne la grande majorité des utilisateurs.

Toute la magie d'Eclipse tient au fait que cette plateforme constitue un merveilleux assistant pour le programmeur. Les erreurs dans lesquelles `javac` vient s'embourber sont en général traitées par Eclipse avant même la phase de compilation, et si une véritable question se pose, Eclipse propose des solutions. Le programmeur n'a qu'à cliquer sur celle qui lui paraît la plus adaptée – sans effort de réflexion. *Formidable !* s'exclamera le lecteur de cet ouvrage, comme tous les développeurs Java.

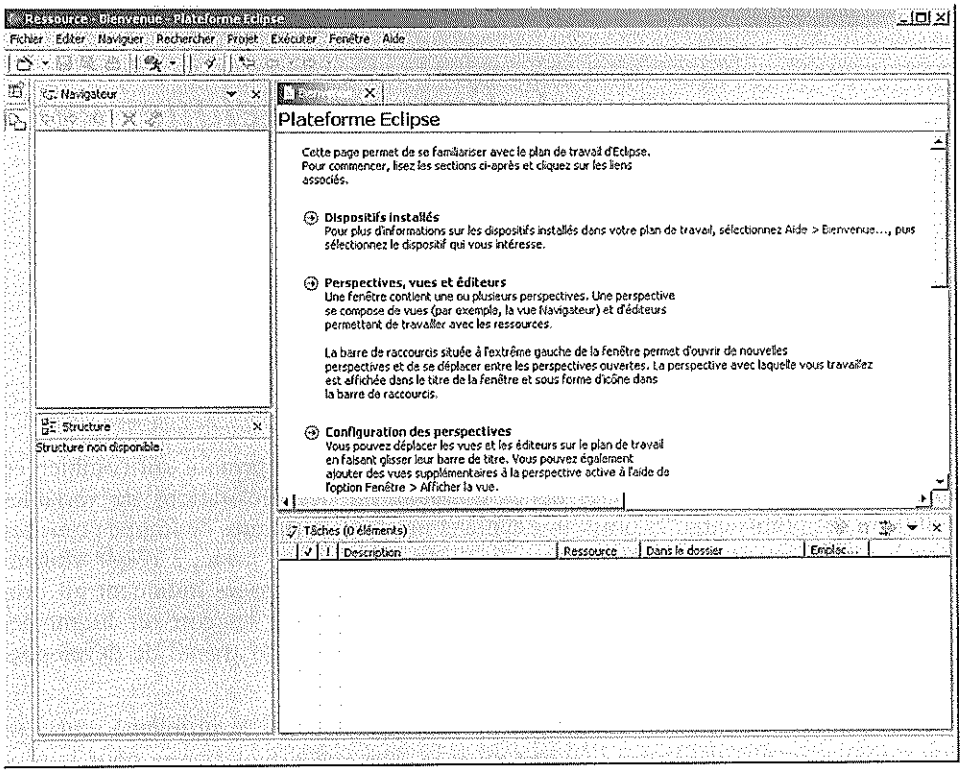


Figure 1-1. Eclipse

Contexte et historique

Eclipse est téléchargeable gratuitement, comme la plupart des EDI Java. Cependant, Eclipse a un avantage notable sur ses concurrents : la puissance d'IBM, qui a financé le développement de cet EDI à hauteur de 40 millions de dollars. La première version (version 1.0) a été diffusée en novembre 2001 et a connu un succès populaire grandissant (même si – comme la plupart des outils de développement – beaucoup de controverses sur ses défauts ont éclaté).

Avec le temps, Eclipse a évolué et s'est amélioré ; la dernière version (2.1.1) recueille à présent beaucoup de louanges. En fait, Eclipse est devenu si populaire qu'à la sortie de la version 2.1, les serveurs du site *http://www.eclipse.org* étaient sollicités à tel point qu'il était pratiquement impossible d'en télécharger une copie dans les jours suivant sa mise à disposition sur le Web.

Eclipse a fait l'objet d'un effort de développement massif de la part d'IBM : les grands gagnants en furent les programmeurs Java. C'est maintenant un projet « open source », sur lequel travaille encore largement IBM, même si ce projet est géré sous la forme d'un consortium de développement nommé eclipse.org. On peut voir la page Web de ce consortium, *http://www.eclipse.org*, dans la figure 1-2.

Le consortium Eclipse a été créé en novembre 2001 ; à cette époque, il s'est constitué autour de la filiale d'IBM, Object Technologies International (OTI) – qui avait déve-

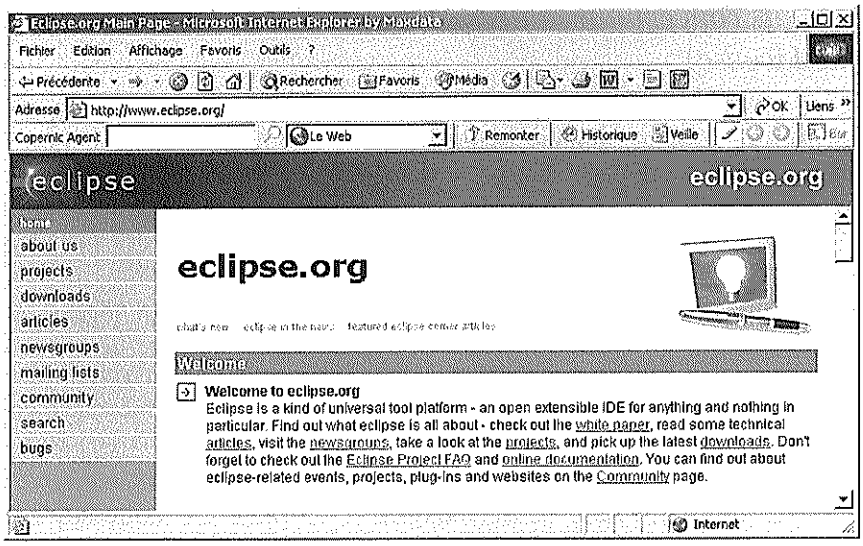


Figure 1-2. Page Web du consortium Eclipse

loppé initialement Eclipse –, de Borland, d'IBM, de MERANT, de QNX Software Systems, de Rational Software3, de Red Hat, de SuSE, de TogetherSoft3 et de Webgain2. Depuis, le consortium s'est étoffé : il comporte maintenant plus de 45 membres, dont Sybase, Hitachi, Oracle, Hewlett-Packard, Intel et bien d'autres.

OTI, racheté par IBM en 1996, est impliqué depuis longtemps dans le développement d'Eclipse. Chargé du développement des produits Visual Age d'IBM, OTI a développé un outil Java écrit en langage Smalltalk, Visual Age pour Java – encore appelé VA4J – qui a connu un succès certain. Eclipse lui-même constitue, dans une certaine mesure, une réécriture et une mise à jour de VA4J en langage Java – la plupart des particularismes de VA4J ont été remplacés au profit de caractéristiques plus standard, et beaucoup d'efforts ont été déployés pour rendre cet outil encore plus puissant. En résumé, Eclipse est relativement nouveau, même s'il a déjà une longue histoire.

L'ensemble du projet Eclipse est composé de trois sous-projets :

- La plateforme Eclipse elle-même, qui constitue l'ossature de base de toute l'application.
- Le JDT (Java Development Toolkit).
- L'environnement de développement de plug-in (PDE, Plug-in Development Environment), qui permet au programmeur de développer ses propres outils pour Eclipse sous la forme de plug-in.

Ces projets sont eux-mêmes subdivisés en sous-projets – le projet JDT, par exemple, est divisé en sous-projets interface utilisateur (UI), noyau et débogueur. Le lecteur intéressé pourra consulter *http://www.eclipse.org/eclipse/*, sachant que cet ouvrage décrit, bien sûr, ces trois projets.

Deux mots concernant la licence d'utilisation (CPL, Common Public License)

Eclipse est un logiciel open source, ce qui peut soulever des difficultés d'interprétation sur le plan légal, pour certains lecteurs. Un logiciel open source permet à ses utilisateurs d'accéder au code source, de le modifier et de le distribuer eux-mêmes. Cependant, les utilisateurs n'ont pas le droit de distribuer le logiciel modifié, sauf si l'utilisateur final en a aussi le droit (cette disposition, appelée *copyleft* dans la communauté open source, est donc différente du copyright traditionnel).

Certaines licences open source mettent en avant le fait que tout logiciel diffusé conjointement avec un autre logiciel de type open source doit être lui-même open source. Cependant, la licence open source concernant Eclipse, la CPL (Common Public License), a été réalisée afin de satisfaire certains intérêts commerciaux. La CPL autorise la diffusion conjointe d'un logiciel open source avec un autre logiciel soumis à des conditions commerciales restrictives de par sa licence.

Si le lecteur de cet ouvrage désire modifier et distribuer de nouvelles versions d'Eclipse, il pourra consulter la CPL sur le site *http://www.opensource.org* (en particulier, *http://www.opensource.org/licenses/cpl.php*). La CPL spécifie, entre autres, que « cette licence a pour but de faciliter l'utilisation commerciale du Programme ».

C'en est terminé des généralités : passons maintenant à l'utilisation effective d'Eclipse.

Installation d'Eclipse

Comment peut-on se procurer Eclipse ? Comment l'installer ? Eclipse peut être téléchargé gratuitement : il suffit de naviguer sur la page *http://www.eclipse.org/downloads* et de sélectionner l'un des sites miroirs permettant le téléchargement. La liste des composants téléchargeables apparaît alors ; on peut dénombrer quatre types essentiels de composants :

Versions diffusables

L'équipe d'Eclipse diffuse ces versions pour un usage général. Ces versions sont, en général, téléchargées en premier lieu. Celles-ci ont été testées en profondeur ; il est très peu probable qu'un bogue sérieux survienne avec ces versions. Si Eclipse faisait l'objet d'une diffusion commerciale, ce sont ces versions qui seraient vendues.

Versions stables

Ces versions sont comparables aux versions bêta. Une version stable est considérée comme fiable et susceptible de devenir une version diffusable. Cependant, certains bogues peuvent subsister. Ces versions permettent de découvrir les nouvelles possibilités offertes par une prochaine version diffusable d'Eclipse.

Versions en phase d'intégration

Ces versions sont constituées de composants ayant fait l'objet de tests unitaires, mais qui n'ont pas encore été validés après leur intégration dans le produit final. Ces versions accéderont au statut de « version stable » une fois les tests d'intégration réalisés.

Versions expérimentales

Ces versions constituent les plus expérimentales parmi celles qui sont rendues publiques. Elles ont été réalisées à titre exploratoire par l'équipe d'Eclipse, mais elles sont livrées sans aucune garantie de bon fonctionnement. Quelques manipulations avec ces versions montrent qu'elles peuvent présenter de gros problèmes.

En général, il convient d'utiliser la version diffusable la plus récente d'Eclipse. Pour l'obtenir, il suffit de cliquer sur le composant téléchargeable le plus récent pour le système d'exploitation concerné : le téléchargement est alors déclenché. En complément, l'utilisateur pourra éventuellement télécharger l'archive « language pack » qui, une fois installée, permet d'obtenir une interface graphique dans la langue de l'utilisateur

Pour en savoir plus sur la dernière version d'Eclipse et sur les versions à venir, le lecteur pourra consulter *http://www.eclipse.org/eclipse/development/main.html*.

L'installation d'Eclipse est très simple : il suffit d'utiliser unzip ou untar, en fonction du système d'exploitation. Ayant téléchargé la version d'Eclipse adaptée à son système d'exploitation, l'utilisateur disposera des fichiers exécutables prêts à l'emploi une fois la décompression terminée.

Les utilisateurs de Windows seront heureux d'apprendre qu'Eclipse n'utilise pas le registre Windows, ce qui rend son installation plus facile et moins sujette à problème.

L'utilisateur appelle Eclipse en lançant l'exécutable ad hoc, par exemple *eclipse.exe*. Lors de son premier lancement, Eclipse peut être amené à compléter l'installation, d'où un court délai d'attente avant le début du travail proprement dit (Eclipse crée divers répertoires de travail). Ainsi, lors du premier lancement d'Eclipse, l'utilisateur doit voir une fenêtre sensiblement identique à celle de la figure 1-1.

Le lecteur notera que Java doit être installé préalablement au lancement d'Eclipse. Si, lors du premier lancement d'Eclipse, une boîte de dialogue apparaît et signale qu'un environnement de développement pour Java ou que le Java Development Kit doivent être installés préalablement (par exemple, « A Java Runtime Environment (JRE) or Java Development Kit (JDK) must be available in order to run Eclipse »), l'utilisateur doit télécharger et installer Java en premier lieu. Java peut être obtenu gratuitement sur le site *http://java.sun.com/j2se/*.
Dans le chapitre suivant, le lecteur verra comment basculer localement d'un environnement de développement pour Java à un autre, tout en restant sous Eclipse (par exemple, pour utiliser un nouveau JDK, que l'utilisateur vient de télécharger, au lieu du JRE fourni par défaut avec de nombreux environnements).

Afin de faciliter l'usage d'Eclipse, l'utilisateur peut associer divers raccourcis à l'exécutable Eclipse. Pour cela, et sous Windows, il suffit de cliquer le bouton droit de la

souris sur le fichier exécutable, dans l'explorateur Windows, de sélectionner « créer un raccourci » dans le menu surgissant, puis de déplacer le nouveau raccourci à l'endroit désiré. Sous Linux ou sous Unix, il suffit d'ajouter le répertoire Eclipse dans la variable système décrivant le chemin d'accès, ou d'utiliser `ln -s` pour créer un lien symbolique vers l'exécutable Eclipse.

Qu'est-ce qu'Eclipse ?

Qu'est-ce qu'Eclipse ? La plupart des programmeurs pensent qu'Eclipse est un EDI pour Java, car le téléchargement d'Eclipse permet d'obtenir aussi l'EDI Java (le JDT, Java Development Toolkit) et l'environnement de développement de plug-ins (PDE, Plug-in Development Environment). Dans ces conditions, il est clair que pour un développeur Java, Eclipse est un EDI Java, puisque c'est cet outil d'Eclipse qu'il utilisera.

En réalité, Éclipse est une *plateforme de développement universelle*, le JDT n'étant qu'un ajout pour Eclipse – il s'agit d'un plug-in. Le logiciel Eclipse est en fait une *plateforme* qui fournit un support pour des outils de développement allant bien au-delà de l'environnement Java fourni à l'occasion du téléchargement. Ces outils sont implémentés sous la forme de plug-in, si bien que la plateforme elle-même ne requiert qu'une bibliothèque logicielle relativement limitée.

La plateforme Eclipse fournit le support logiciel nécessaire au bon fonctionnement des plug-in ; ainsi, le développeur Java utilisera le plug-in JDT diffusé avec Eclipse ; de même, un développeur désirant programmer dans un autre langage utilisera d'autres plug-in, tel que CDT, qui permet de développer du code en langage C/C++. L'installation d'un plug-in est très simple, comme on va le voir – il suffit de le copier dans le répertoire *plugins* et de redémarrer Eclipse. Lors de chaque lancement, Eclipse vérifie les plug-in présents dans ce répertoire, mais ceux-ci ne sont chargés que lorsque cela est nécessaire, afin d'économiser la ressource processeur et l'espace mémoire.

Il convient de bien comprendre qu'Eclipse permet de développer du code dans n'importe quel langage, même si Eclipse a été écrit en Java. Il suffit pour cela de se procurer le plug-in correspondant au langage souhaité. De même, Eclipse a aussi été conçu pour supporter n'importe quelle langue parlée – il est très facile de changer la langue utilisée par Eclipse. Pour ce faire, il faut utiliser le même mécanisme de plug-in que pour un langage informatique, à ceci près que les langues sont gérées au moyen de *fragments de plug-in*. OTI propose une bibliothèque dédiée aux langues incluant, entre autres, le japonais, le coréen, l'allemand, le français, l'italien,le portugais, l'espagnol et même le chinois traditionnel et simplifié.

La plateforme Eclipse

La plateforme Eclipse est constituée d'un certain nombre de composants : le noyau de la plateforme, le plan de travail, l'espace de travail, le support d'équipe et le serveur d'aide. La figure 1-3 présente une vision synthétique de la plateforme.

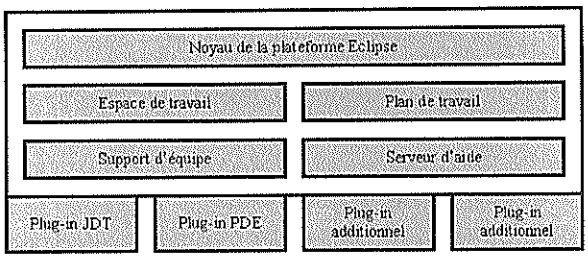


Figure 1-3. L'architecture d'Eclipse

Le noyau de la plateforme

Le rôle du noyau est de lancer au bon moment les outils logiciels constituant Eclipse ; le noyau charge aussi les plug-in lorsque cela est nécessaire. Lorsqu'Eclipse est lancé, le noyau est le premier composant à être exécuté : il charge tous les plug-in nécessaires au bon fonctionnement d'Eclipse – le plan de travail, par exemple – , et qui sont parfois faussement assimilés à Eclipse lui-même.

Le plan de travail d'Eclipse

La figure 1-1 présente l'apparence visuelle du plan de travail d'Eclipse – celui-ci représente l'interface graphique de base proposée par Eclipse à ses utilisateurs. Ce plan de travail comporte toutes sortes de barres d'outils et de menus destinés à faciliter le travail du développeur, ainsi qu'un certain nombre d'éléments graphiques, dans un environnement multi-fenêtre (voir la figure 1-1).

Si l'on excepte le noyau, le plan de travail d'Eclipse est le composant le plus central d'Eclipse. Lorsque Eclipse est lancé, le plan de travail se contente d'afficher un message de bienvenue, le choix d'un EDI spécifique, tel que le JDT, venant après. Lorsqu'un autre outil, tel que le JDT, est lancé, c'est lui qui prend la main.

Le plan de travail a l'apparence d'une application native, destinée au système d'exploitation de l'utilisateur, ce qui constitue tout à la fois une spécificité d'Eclipse et un sujet de controverse. Le plan de travail lui-même – i.e. l'interface graphique utilisateur d'Eclipse – a été construit sur la base du SWT (Standard Widget Toolkit) propre à Eclipse et JFace (qui a été conçue en s'appuyant sur SWT). Le SWT fait appel aux bibliothèques logicielles natives du système d'exploitation, afin de fournir le look-and-feel d'une application native du système d'exploitation. Cette approche va à l'encontre des pratiques historiques liées au développement d'applications Java, même de celles qui font appel a Swing.

SWT doit faire l'objet d'un portage vers tout système d'exploitation supportant Eclipse, ce qui a suscité un vif débat dans la communauté Eclipse, nombre d'utilisateurs affirmant qu'Eclipse devait être, tout comme Java, totalement indépendant de tout système d'exploitation. Quoi qu'il en soit, les concepteurs d'Eclipse ont décidé de poursuivre dans la voie choisie, si bien qu'Eclipse a été porté sur la plupart des principaux systèmes d'exploitation incluant, entre autres, Windows, Solaris, Mac OS X, Linux/Motif, Linux/GTK2 et HP-UX.

Dans cet ouvrage, on verra en fait comment utiliser SWT et JFace pour créer des applications Java au look-and-feel totalement natif. SWT supporte les contrôles et les éléments graphiques de base, JFace fournissant une palette considérablement étendue de composants. Eclipse n'est pas seulement construit sur la base de SWT et de JFace, mais il permet aussi au développeur de les utiliser tels quels.

L'espace de travail d'Eclipse

L'espace de travail d'Eclipse gère toutes les ressources nécessaires au travail du développeur – i.e. tout ce qui est stocké sur disque dur et tout ce qui permet la connexion à d'autres machines. Tout code écrit sous Eclipse fait partie d'un *projet* Eclipse. Chaque projet se voit attribuer un dossier propre dans le répertoire de travail d'Eclipse, ce qui facilite le suivi et la traçabilité des différents projets créés sous Eclipse. Chaque projet comporte lui-même un certain nombre de sous-dossiers. En général, tous les dossiers d'un projet sont des sous-dossiers du dossier principal du projet, même si cela n'est pas une obligation – il est possible d'inclure dans un projet des dossiers se trouvant n'importe où sur l'ordinateur utilisé ou sur un autre ordinateur accessible par le réseau.

Lors de la phase de développement du code, l'espace de travail se charge de gérer toutes les ressources associées au projet, en particulier tous les fichiers inclus dans le projet. Il sauvegarde les modifications effectuées dans ces ressources, mémorisant au passage l'historique des changements de chaque ressource ; ceci permet au programmeur de revenir sur ces changements selon ses besoins. L'espace de travail informe les plug-in de ces modifications.

Le fait de stocker tous les projets Eclipse au sein d'un même répertoire a ses avantages. Par exemple, lorsqu'une nouvelle version d'Eclipse est installée, il suffit au programmeur de copier le répertoire de l'espace de travail de l'ancienne version vers celui de la nouvelle (le lecteur vérifiera tout de même dans la documentation qu'agir de la sorte ne pose pas de problème – par exemple, il n'y a pas de compatibilité ascendante entre les projets de la version 1.0 d'Eclipse et ceux de la version 2.0 ou d'une version postérieure. Si un problème se pose, il est toujours possible d'effacer la nouvelle version et de réinstaller Eclipse au moyen de unzip ou de untar).

Le support d'équipe

Le support d'équipe est le plug-in en charge du contrôle de version du code dans Eclipse. Avec ce contrôle, le code du programme est enregistré dans une archive ou extrait de celle-ci, afin de pouvoir reconstituer par la suite l'historique des modifications dans ce code. Cette tâche est réalisée de telle façon qu'il n'y ait ni écrasement ni perte des modifications effectuées par les autres membres de l'équipe, même lorsqu'ils travaillent sur différentes versions du code en même temps.

Ce composant fonctionne comme un client CVS (Concurrent Versions System) interagissant avec un serveur CVS. Que le lecteur non familier de CVS se rassure : il trouvera plus loin un exemple de suivi des modifications dans un code au moyen de

CVS (chapitre 4). L'utilisation du support d'équipe assure un contrôle efficace des différentes versions d'un code, chose extrêmement utile dans le cadre d'un travail en équipe.

Le serveur d'aide

Le serveur d'aide fournit, comme son nom l'indique, des informations venant aider le programmeur. Il s'agit en réalité d'un système de documentation extensible destiné à fournir une aide ; les plug-in peuvent fournir une documentation HTML au format XML, qui indique ainsi comment naviguer dans l'aide.

Ainsi se termine le tour d'horizon des principaux composants d'Eclipse. Pour *utiliser* Eclipse, il convient de connaître quelques concepts supplémentaires : les vues et les perspectives.

Vues et perspectives

Le plan de travail est un environnement multi-fenêtre, chaque fenêtre présentant l'état des développements informatiques selon un certain point de vue : une telle fenêtre est encore appelée *vue*. Ainsi, une fenêtre peut montrer l'ensemble des classes Java invoquées dans un projet ; une autre permet de naviguer d'un projet à un autre. Par exemple, la figure 1-1 montre la vue Navigateur dans le coin supérieur gauche d'Eclipse – dans celle-ci, on peut voir tous les projets développés par le programmeur, qui peut basculer de l'un à l'autre à tout moment.

Afin de gérer au mieux l'espace visuel à l'écran, Eclipse propose un système d'onglets permettant de basculer d'une vue à l'autre.

Pour ouvrir à nouveau une vue fermée par erreur, il suffit de cliquer l'option de menu Fenêtre → Afficher la vue, et de sélectionner la vue concernée.

L'éditeur est une fenêtre spéciale qui apparaît en général au centre du plan de travail. Les documents (code ou ressources) ouverts par l'utilisateur apparaissent dans l'éditeur. Celui-ci est choisi par Eclipse en fonction du type de document ouvert : si le document est un fichier de code Java, ce sera l'éditeur Java ; si le document est un GUI développé au moyen d'un plug-in, c'est l'éditeur du plug-in qui sera ouvert ; etc. Il est même possible d'ouvrir des documents pour Microsoft Word : Eclipse ouvre alors une fenêtre MS Word au moyen de Windows OLE (Object Linking and Embedding). Dans la figure 1-1, l'espace normalement réservé aux éditeurs ne montre que le message de bienvenue d'Eclipse.

La fenêtre d'édition est le lieu principal pour le développement de code ; c'est dans cette fenêtre que le code est tapé ; comme on va le voir, le JDT possède un éditeur somptueux et riche en possibilités, telles que la vérification de syntaxe, mise en surbrillance de code, etc. Plusieurs éditeurs peuvent être ouverts simultanément et insérés dans le système d'onglets : la sélection de l'un d'entre eux est effectuée ainsi (ou au moyen de l'option de menu Fenêtre → Basculement vers un éditeur..., qui affiche la liste des éditeurs ouverts). La fermeture d'un éditeur est effectuée en cliquant le X de l'onglet associé (ou au moyen de l'option de menu Fenêtre → Masquer les éditeurs, qui est alors remplacée dans le menu par l'option Fenêtre → Affi-

cher les éditeurs). En résumé : les vues présentent vos projets selon divers points de vue, et les éditeurs permettent de développer du code et des ressources.

Eclipse introduit un dernier concept : les *perspectives*. En général, le programmeur ne décide pas lui-même des vues et des éditeurs associés à un projet ; le choix de ceux-ci est conditionné dans le cadre de perspectives, qui sont des groupes de choix liés à un type de développement : par exemple, c'est la perspective Java (et donc les éditeurs et les vues correspondantes) qui sera choisie par Eclipse si le programmeur crée une application Java ; de même, c'est la perspective Débogage qui sera choisie par Eclipse si le programmeur débogue un programme Java. Le programmeur peut néanmoins créer ses propres perspectives.

Les perspectives permettent de spécifier des ensembles prédéfinis de vues et d'éditeurs automatiquement appelés dans un contexte donné. Il existe par exemple une perspective Java, que l'on peut sélectionner au moyen de l'option de menu Fenêtre → Ouvrir la perspective, qui affiche toutes les perspectives disponibles et installées, et en cliquant l'option Fenêtre → Ouvrir la perspective → Java. On obtient ainsi la perspective Java présentée dans la figure 1-4.

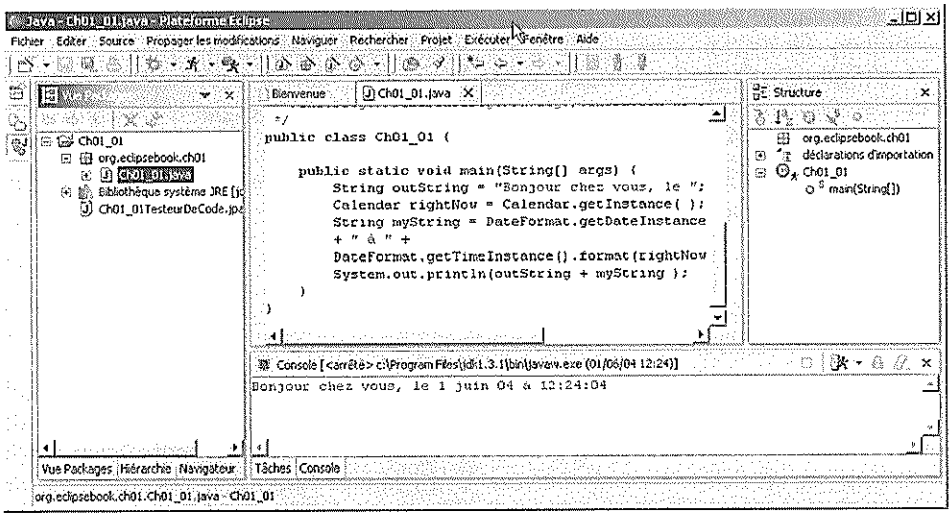


Figure 1-4. La perspective Java

Cette perspective propose un éditeur apparaissant au centre de la fenêtre d'Eclipse, et qui permet de développer du code Java. Cette perspective propose en outre diverses vues. Par exemple, la vue Packages, visible à gauche de la fenêtre, permet de voir les paquetages et les classes Java d'un projet et de naviguer parmi eux. Lorqu'une application Java est déboguée, le plan de travail s'appuie sur la perspective Débogage, visible dans la figure 1-5. Cette perspective affiche le code en cours de débogage, la ligne en cours d'exécution, les valeurs des variables et des expressions que l'utilisateur veut avoir sous les yeux, et d'autres renseignements utiles, visualisés dans d'autres vues.

La plupart du temps, les notions de vues, d'éditeurs et de perspectives sont transparentes, voire triviales pour l'utilisateur, chevronné ou non, manipulant quoti-

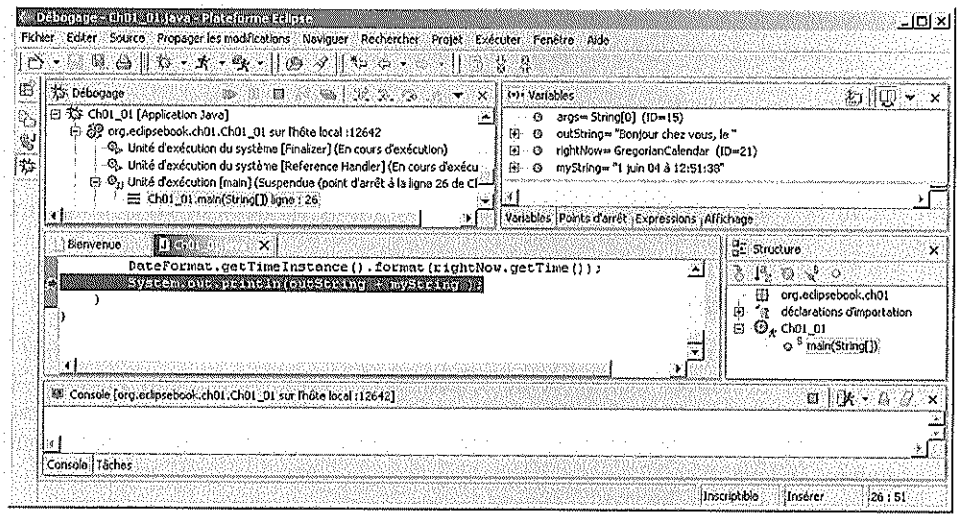


Figure 1-5. La perspective Débogage

diennement Eclipse. Ces notions sont cependant très importantes dans le monde d'Eclipse : on y reviendra souvent tout au long de cet ouvrage.

Les concepts essentiels liés à Eclipse viennent d'être présentés. On peut maintenant commencer à travailler avec Eclipse. Bien qu'Eclipse soit une plateforme de développement universelle, sa popularité vient essentiellement de son utilisation par la communauté Java : la présentation qui suit s'appuie donc sur le développement d'un programme Java. Lorsqu'Eclipse est lancé pour la première fois, c'est la perspective Ressource qui est appelée : c'est la perspective par défaut pour développer des ressources au sens large. Mais Eclipse va basculer vers la perspective Java dès que l'on va commencer à écrire du code Java.

Travailler sous Eclipse

L'utilisation d'Eclipse s'appuie sur le JDT (Java Development Tools), un ensemble de six plug-in parfaitement intégrés dans Eclipse et destinés au développement d'applications Java sous Eclipse. Le JDT apporte une réelle amélioration en termes d'EDI, même pour les programmeurs Java de longue date ; lorsque le lecteur aura compris l'utilisation du JDT, il se demandera pourquoi il a fallu attendre aussi longtemps pour que cet outil soit une réalité.

Utiliser Eclipse, c'est programmer du code. On va donc utiliser le JDT pour réaliser l'application présentée dans l'exemple 1-1. Cette application affiche tout simplement le message « Bonjour chez vous. » à l'écran.

Exemple 1-1. L'exemple Ch01_01.java

```
public class Ch01_01
{
    public static void main(String[] args)
    {
```

Exemple 1-1. L'exemple Ch01_01.java (suite)

```
            System.out.println("Bonjour chez vous.");
        }
    }
```

Comment peut-on créer cette application avec Eclipse, et comment Eclipse rend-il le travail plus facile ? Voici la réponse à ces deux questions. Sous Eclipse, tout code Java doit être intégré dans un projet : la première chose à faire est donc de créer un projet Java.

Création d'un projet Java

Pour invoquer la perspective Java et saisir le code de ce premier exemple, *Ch01_01.java*, il suffit de lancer Eclipse et de sélectionner l'option de menu Fenêtre → Ouvrir la perspective → Java : le JDT est alors ouvert avec la perspective Java, comme on peut le voir dans la figure 1-6. C'est cette perspective qui sera utilisée pour tout développement Java.

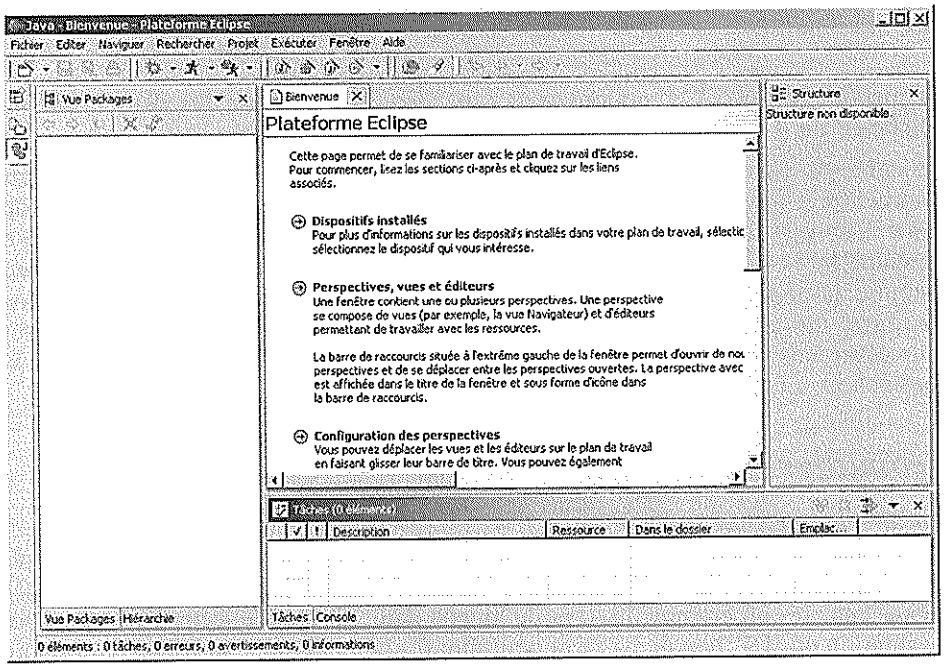


Figure 1-6. La perspective Java d'Eclipse

Il est bon de connaître la perspective Java avant de l'utiliser. Dans sa partie supérieure, on trouve la barre de menus et la barre d'outils traditionnelle, qui proposent des options spécifiques de la perspective Java, qui deviendront familières dans les prochaines pages.

La partie gauche d'Eclipse présente la vue Packages et la vue Hiérarchie, accessibles au moyen d'onglets. La vue Packages permet d'avoir une vision d'ensemble du paquetage développé et de naviguer dans les différents projets Java en cours : ainsi, il

est possible d'accéder rapidement aux fichiers de code. La vue Hiérarchie permet d'examiner la hiérarchie des types – en sélectionnant une classe ou un objet dans du code en cours d'édition, en cliquant le bouton droit de la souris et en sélectionnant l'option de menu contextuel Ouvrir la hiérarchie des types, on obtient la hiérarchie de cet objet (au sens de l'héritage). Il est possible de cliquer sur un élément de cet arbre généalogique afin d'accéder à tous ses membres. Cette vue est d'une grande aide lorsque le programmeur recherche l'existence ou le prototype d'une méthode

La vue Structure, qui se trouve dans la partie droite de la fenêtre d'Eclipse, propose une vision hiérarchique et structurée du contenu du fichier ouvert dans la fenêtre d'édition ; cette vue permet d'accéder très rapidement aux différents éléments codés dans le fichier. Cet outil est particulièrement apprécié des programmeurs habitués à écrire de longs fichiers de code Java, car il offre une vision synthétique des principales sections d'un fichier de code (on reviendra sur cette vue dans le prochain chapitre).

Dans la partie inférieure de la fenêtre d'Eclipse se trouvent la vue Tâches et la vue Console, qui sont accessibles au moyen d'onglets. La vue Tâches affiche les différents tâches en attente, comme par exemple les erreurs découvertes par le compilateur et qui nécessitent une action du programmeur ; la vue Console montre le résultat de l'exécution du programme sur la console de sortie – notre exemple test affichera donc son message dans la vue Console.

Les éditeurs sont empilés au centre de la fenêtre d'Eclipse ; ils sont accessibles au moyen d'onglets situés dans le coin supérieur gauche de la zone d'édition. Les éditeurs de code JDT sont extrêmement puissants et vont largement au-delà de la simple frappe de texte. Ils contiennent toutes sortes d'outils cachés dont la plupart sont très discrets jusqu'à ce que le programmeur les utilise.

Pour créer un nouveau projet Java, dans la perspective Java, il suffit de sélectionner l'option de menu Fichier → Nouveau → Projet... (ou de cliquer le bouton droit de la souris dans la vue Packages et de sélectionner l'option de menu contextuel Nouveau → Projet) ; on obtient ainsi la boîte de dialogue Nouveau Projet, comme le montre la figure 1-7.

Pour passer à l'étape suivante, il suffit de sélectionner les options Java et Projet Java, et de cliquer Suivant. On tape ensuite le nom du nouveau projet, `Ch01_01`, dans la zone de texte intitulée Nom du Projet ; en cliquant à nouveau Suivant, on arrive à l'étape présentée dans la figure 1-8. Comme on peut le voir dans celle-ci, on est sur le point de créer le nouveau projet `Ch01_01` et son répertoire associé. On peut utiliser l'onglet Projets pour inclure d'autres projets nécessaires à la construction de l'exécutable, possibilité qui sera exploitée dans des exemples plus étoffés. L'onglet Bibliothèques permet d'explorer les bibliothèques et les fichiers JAR à inclure dans le projet ; par défaut, seule la bibliothèque système JRE est incluse. L'onglet Ordre et exportation permet au programmeur de compiler l'ordre des chemins de classe et d'exporter son code afin de le rendre accessible par d'autres projets. Dans le cas présent, il suffit de cliquer Fin pour créer le nouveau projet `Ch01_01`.

Cette action ajoute le nouveau projet `Ch01_01` dans la vue Packages, comme on peut le voir dans la figure 1-9. Le projet est représenté sous la forme d'un dossier dans l'espace de travail et, en l'état actuel, ce dossier contient les seules bibliothèques incluses dans le chemin de classe, la bibliothèque système JRE dans le cas présent.

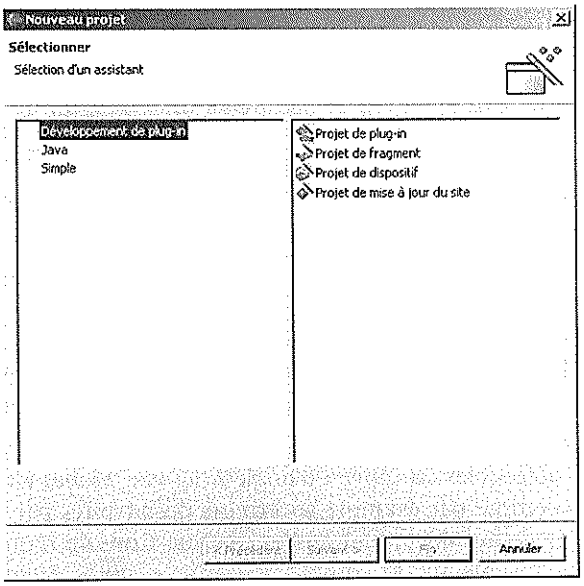


Figure 1-7. Boîte dialogue Nouveau Projet, première étape

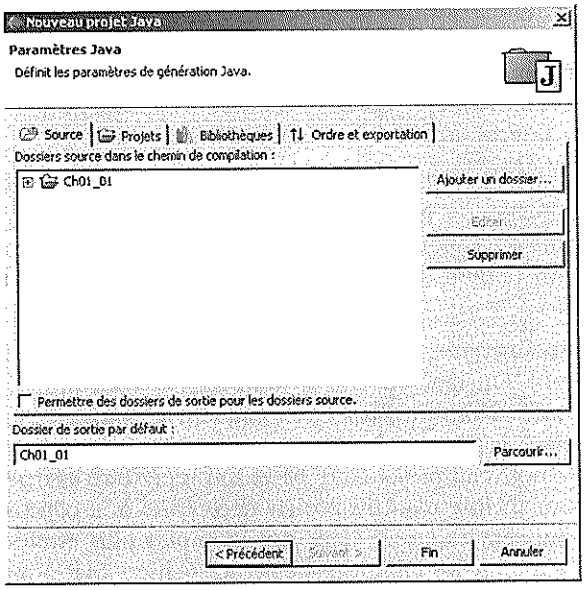


Figure 1-8. Boîte dialogue Nouveau Projet, troisième étape

Des projets comme celui-ci permettent d'organiser les fichiers, les classes, les bibliothèques et les exports. Pour le moment, aucune ligne de code n'a été écrite : la prochaine étape consiste donc à écrire une classe publique Java.

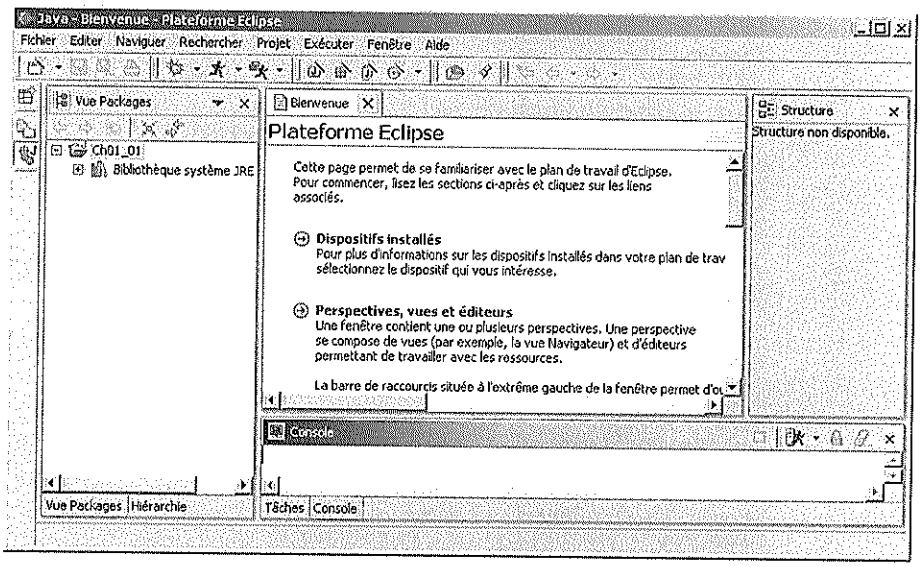


Figure 1-9. un nouveau projet dans la vue Packages

Création d'une classe Java

Dans notre exemple, la classe publique s'appelle `Ch01_01` ; Eclipse stocke les classes publiques comme celle-ci à raison d'une classe par fichier. Il existe plusieurs façons de créer de nouvelles classes sous Eclipse : on peut utiliser, dans la barre d'outils, le bouton contenant un C apparaissant sur un petit disque vert ; on peut utiliser l'option de menu Fichier → Nouveau → Classe ; on peut aussi cliquer le bouton droit de la souris sur un projet, dans la vue Packages, et sélectionner l'option de menu contextuel Nouveau → Classe. Dans tous les cas, on voit apparaître la boîte de dialogue Nouvelle Classe Java, présentée dans la figure 1-10.

On notera les différentes options présentées dans cette boîte de dialogue. On peut spécifier certains modificateurs de classe – `public`, `private`, ou `protected` ; on peut rendre la classe `abstract` ou `final` ; on peut spécifier sa classe mère (`java.lang.Object` est la classe mère par défaut) ; on peut enfin spécifier les éventuelles interfaces qui sont implémentées par la classe. Les classes présentées dans tous les exemples de ce livre seront intégrées dans des paquetages Java pour éviter tout conflit de code ; pour nommer un paquetage, on conviendra d'utiliser le nom du chapitre contenant le code concerné, par exemple `org.eclipsebook.ch01`. Pour ce faire, dans le cas présent, il suffit de taper le nom de la nouvelle classe, `Ch01_01`, dans la zone de texte intitulée Nom, de taper le nom du nouveau paquetage pour cette classe, `org.eclipsebook.ch01`, dans la zone de texte intitulée Package, et de cliquer Fin pour valider tous les autres choix par défaut. On notera en particulier qu'à la question « quels raccords de méthode voulez-vous créer ? », le choix « public static void main(String[] args) » reste coché. Agir ainsi indique qu'Eclipse doit créer automatiquement une méthode `main` vide.

Lorsque le bouton Fin est cliqué, la nouvelle classe publique `Ch01_01` est créée, comme on peut le voir dans la figure 1-11 ; on notera la présence de la ligne package

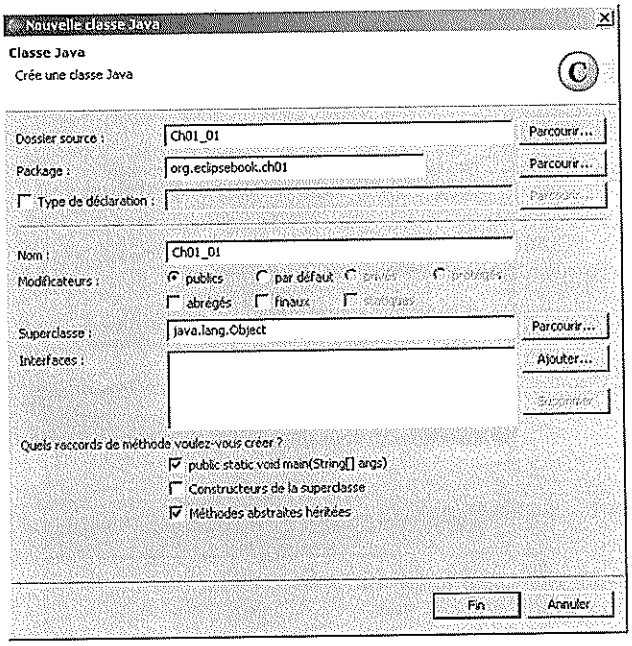


Figure 1-10. Création d'une nouvelle classe Java

`org.eclipsebook.ch01 ;`, qui signale que ce paquetage vient d'être créé : la nouvelle classe sera stockée dans le fichier *Ch01_01.java*, lui-même sauvegardé dans le répertoire *workspace\Ch01_01*.

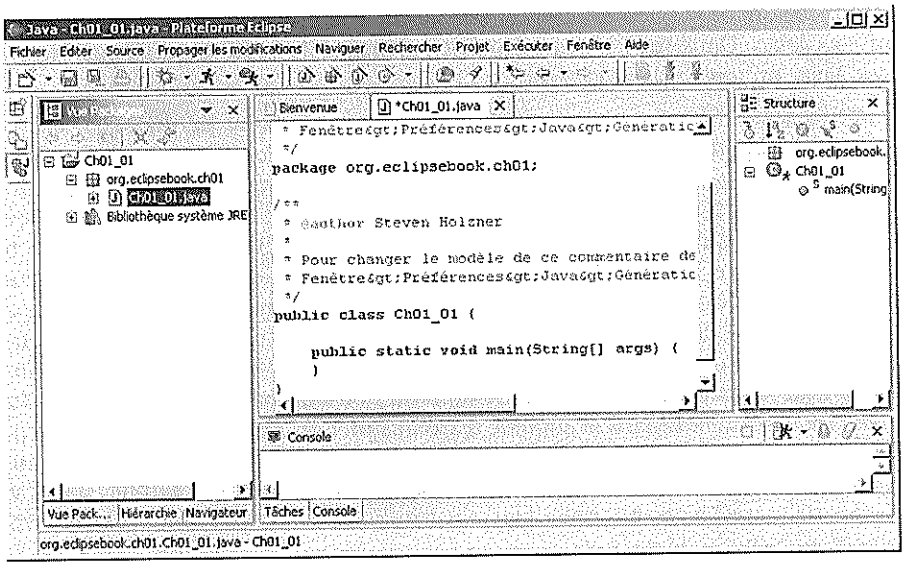


Figure 1-11. Une nouvelle classe Java dans Eclipse

Jusqu'à présent, tout va bien ; on peut voir qu'Eclipse a automatiquement inclus une méthode `main` dans cette nouvelle classe. Nous allons maintenant saisir notre propre code.

Utilisation de l'assistant de code

On peut taper du code grâce à l'éditeur de JDT aussi bien qu'avec tout autre éditeur, mais le JDT propose de nombreux outils spécifiques. Par exemple, il comporte un utilitaire nommé *assistant de code*, qui complète automatiquement le code tapé par le programmeur lorsque des mots-clés sont reconnus par l'assistant.

Imaginons par exemple que le programmeur souhaite taper `System.out.println("Bonjour chez vous.");` dans la méthode `main`. Pour voir l'assistant de code en action, il faut amener le curseur de la souris dans le corps de la méthode `main`, taper `System.` et attendre. L'assistant de codage va afficher automatiquement les diverses classes contenues dans l'espace de nommage `System`, comme on peut le voir dans la figure 1-12.

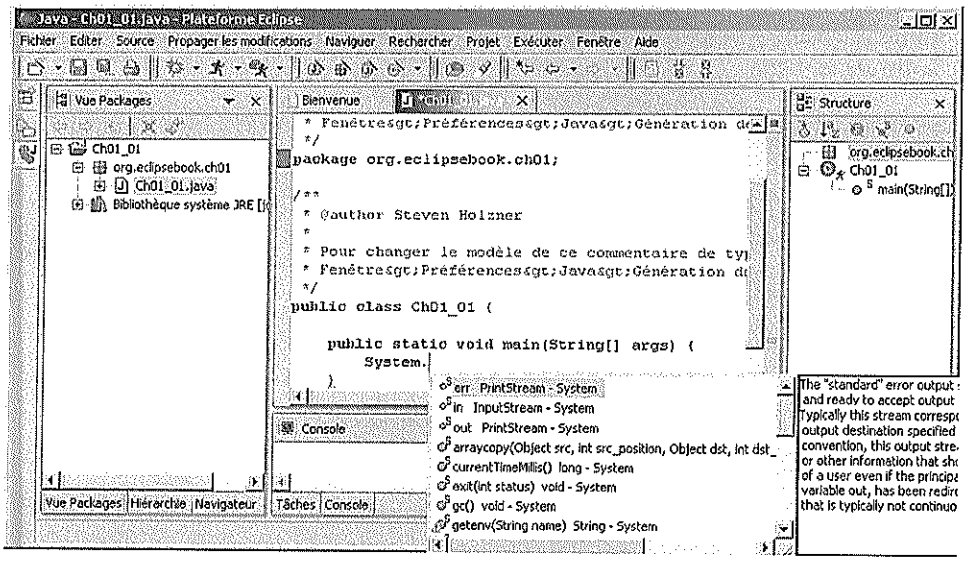


Figure 1-12. Utilisation de l'assistant de code pour écrire l'appel d'une méthode

Lorsque l'objet `out` est sélectionné dans la liste des classes possibles, l'assistant de codage rappelle le rôle de cette classe, comme on peut le voir sur la partie droite de la figure. En double-cliquant sur `out` dans la liste fournie par l'assistant de code, le programmeur voit celui-ci insérer ce texte dans le code ; en tapant un point, afin d'obtenir `System.out.`, et en attendant de nouveau, le programmeur verra l'assistant de code lui proposer les différentes méthodes de la classe `out`. En double-cliquant sur la suggestion `println(Stringx)`, le programmeur verra l'assistant de code insérer automatiquement ce code dans la méthode `main` :

```
public class Ch01_01
{
    public static void main(String[] args)
```

```
    {
        System.out.println()
    }
}
```

On va maintenant compléter le code en tapant le message à afficher, "Bonjour chez vous." (on notera que l'assistant de code ajoute automatiquement le guillemet fermant) :

```
public class Ch01_01
{
    public static void main(String[] args)
    {
        System.out.println("Bonjour chez vous.")
    }
}
```

Cependant, Eclipse affiche cette nouvelle ligne de code en la soulignant d'une ligne rouge ondulée, ce qui indique l'existence d'un problème. Pour en savoir plus, il suffit de placer le curseur de la souris sur cette ligne ; une bulle d'aide surgit alors à l'écran, comme on peut le voir dans la figure 1-13 : il apparaît qu'un point-virgule manque à la fin de la ligne.

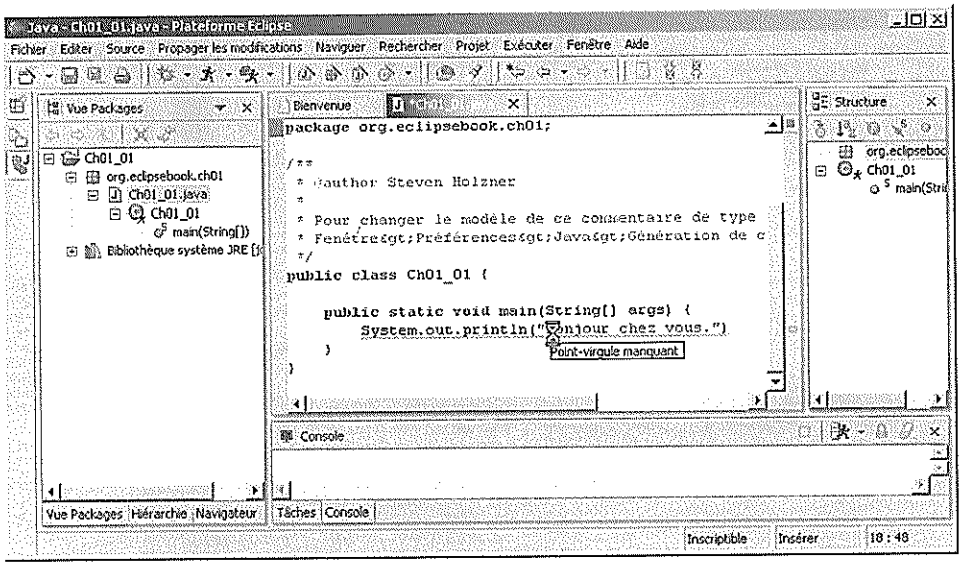


Figure 1-13. mise en évidence d'une erreur

L'insertion du point-virgule manquant en fin de ligne fait disparaître la ligne rouge :

```
public class Ch01_01
{
    public static void main(String[] args)
    {
        System.out.println("Bonjour chez vous.");
    }
}
```

Le code est alors complet. Dans la figure 1-13, on remarquera que la vue Packages présente une vue synthétique du projet (la classe publique `Ch01_01` et la méthode `main` de cette classe). Ainsi, la vue Packages permet d'accéder à tous les éléments constitutifs d'un projet. Il suffit pour cela de double-cliquer sur un élément, dans cette vue, pour que celui-ci apparaisse dans l'éditeur de code.

Une autre méthode pratique permettant d'accéder aux membres d'une classe (ou d'un objet) est de sélectionner le nom de celle-ci dans l'éditeur de code, de cliquer le bouton droit de la souris sur celle-ci, et de sélectionner l'option Ouvrir la hiérarchie des types dans le menu contextuel. On obtient ainsi la hiérarchie des types complète de cette classe (ou de cet objet), tous ses membres et leur type, les prototypes des méthodes, etc. De plus, le code contenant la définition de la classe sélectionnée apparaît dans l'éditeur de code (sauf si le code est inaccessible, en particulier lorsqu'il est dans un fichier JAR – par exemple, la classe `System.out` est définie dans `rt.jar` et son code n'est pas accessible – bien qu'il existe maintenant des méthodes permettant d'attacher du code source aux fichiers JAR).

Comme on peut le voir dans cet exemple, le codage a été facilité grâce à l'assistant de code, qui connaît les membres de la classe `System.out` et permet au programmeur de sélectionner parmi eux celui dont il a besoin. L'assistant de code apparaît automatiquement lorsque le caractère point (.) est tapé. Il apparaît aussi à tout moment en cours de saisie, sur demande du programmeur – il suffit pour cela de taper Ctrl+Espace.

On peut aussi désactiver l'assistant de code. Pour le configurer selon ses besoins, il suffit de sélectionner Fenêtre → Préférences, puis Java et Editeur dans la partie gauche de la boîte de dialogue Préférences et de cliquer l'onglet Assistant de code.

Chaque fois que du code est édité dans un fichier, comme on l'a fait ici pour *Ch01_01.java*, un astérisque apparaît devant le nom de ce fichier (voir la figure 1-13). Cet astérisque indique que le fichier a été modifié mais que ces modifications n'ont pas encore été sauvegardées. Pour sauvegarder un fichier, plusieurs méthodes sont possibles : on peut cliquer l'icône représentant une disquette dans la barre d'outils (action équivalente à Sauvegarder) ; dans cette même barre d'outils, on peut cliquer l'icône représentant une disquette suivie de trois points de suspension (...) (action équivalente à Sauvegarder sous...) ; on peut aussi cliquer le bouton droit de la souris sur le code lui-même et sélectionner l'option de menu contextuel Sauvegarder ; on peut enfin sélectionner les options de menu Sauvegarder, Sauvegarder sous... ou Sauvegarder tout dans le menu Fichier.

Maintenant que le code a été saisi et sauvegardé sur le disque, on va l'exécuter.

Exécution du code

Un examen très attentif de la vue Packages (voir la figure 1-13) permet de constater que l'icône représentant un C pour notre classe `Ch01_01` comporte aussi un petit personnage qui court. Éclipse indique de cette façon que la classe est *exécutable*, parce

qu'elle possède une méthode `main`. Pour exécuter ce code et observer son effet, on peut sélectionner l'option de menu Exécuter → Exécuter en tant que → Application Java, ou bien ouvrir le menu associé à l'icône représentant un personnage qui court, dans la barre d'outils, et sélectionner l'option de menu Exécuter en tant que → Application Java. Cela déclenche l'exécution du code (Eclipse propose toujours de sauvegarder celui-ci s'il ne l'a pas encore été), et la méthode `System.out.println` affiche le fameux message « Bonjour chez vous. » dans la vue Console, dans la partie inférieure de la perspective Java (voir la figure 1-14).

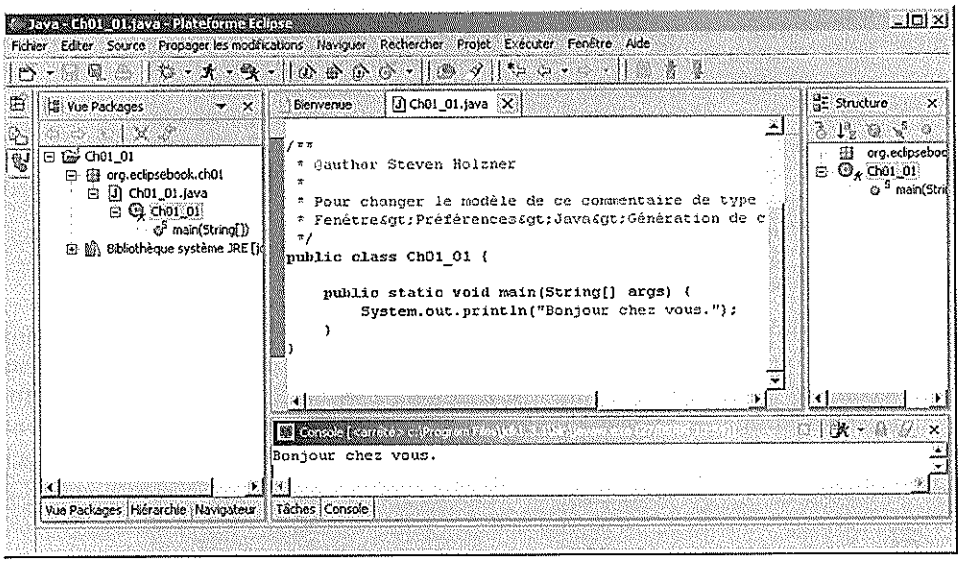


Figure 1-14. Exécution de l'exemple Ch01_01

Félicitations ! Vous êtes devenu un développeur Eclipse.

Utilisation du testeur de code

Il existe une autre façon d'exécuter du code dans un projet Java, sans disposer d'une méthode `main` : il faut pour cela utiliser une *page de testeur de code*. Les pages de testeur de code permettent d'exécuter du code, même partiellement, à la volée – cette possibilité n'est pas essentielle, même si elle peut s'avérer très utile durant la phase de développement.

Pour créer une page de testeur de code, il faut sélectionner l'option de menu Fichier → Nouveau → Page de testeur de code, taper le nom de la page (ici, `Ch01_01TesteurDeCode`) dans la zone de texte intitulée Nom du fichier, et cliquer Fin pour créer la nouvelle page de testeur de code, nommée ici *Ch01_01TesteurDeCode.jpage*. Cette nouvelle page apparaît dans la vue Packages ; elle est ouverte automatiquement dans la vue Editeur, comme on peut le voir dans la figure 1-15. Il est possible de taper du code ou de courts extraits de code à exécuter dans cette page, chose très utile lorsque le volume de code croît et que le programmeur veut juste en tester une partie. Pour exécuter notre exemple, il suffit de saisir le code dans la page de testeur de code (voir la figure 1-15) – on notera qu'il faut inclure le nom du paquetage lorsqu'on fait

appel à la méthode main dans le code :

```
String[] args = {};
org.eclipsebook.ch01.Ch01_01.main(args);
```

Pour indiquer à la page de testeur le code à exécuter, il suffit de sélectionner le code

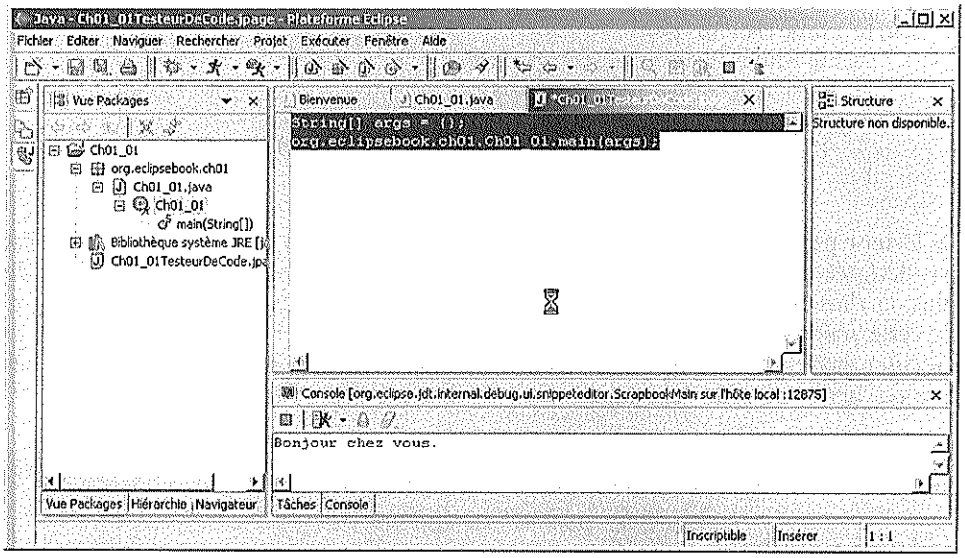


Figure 1-15. Utilisation d'une page de testeur de code

concerné (voir la figure 1-15), de cliquer le bouton droit de la souris sur celui-ci et de sélectionner le choix de menu contextuel Exécuter (on peut aussi sélectionner le choix de menu Exécuter → Exécuter). Le résultat apparaît dans la vue Console, comme précédemment (voir la figure 1-15). De cette façon, on peut exécuter du code Java (ou de courts extraits de code) grâce à la page de testeur de code, et voir ainsi l'effet de celui-ci. Pour fermer la page de testeur de code, on peut cliquer sur le bouton X de l'onglet correspondant à l'éditeur associé, dans la zone centrale de la fenêtre d'Eclipse.

Pour examiner la valeur retournée par le code exécuté dans le testeur de code, il suffit de sélectionner le choix de menu Afficher (au lieu de Exécuter), ce qui permet de tester plus efficacement le bon fonctionnement du code. On notera qu'il est possible de définir des importations à l'intérieur de la page de testeur de code, afin d'importer des paquetages selon les besoins ; pour cela, on clique le bouton droit de la souris sur le code contenu dans le testeur de code, dans la vue Editeur, et l'on sélectionne le choix de menu contextuel Définir les importations.

Comme on vient de le voir, le JDT fournit une aide très appréciable, mais il ne s'agit là que d'un exemple. Il peut également aider le programmeur si son code comporte une erreur.

Utilisation du correctif rapide (fonction Quick Fix)

Le correctif rapide Quick Fix est un outil permettant au JDT de formuler des propositions pour résoudre des erreurs simples, chose longtemps attendue dans la communauté Java. Supposons par exemple que l'on modifie le code de `Ch01_01` pour afficher le message « Bonjour chez vous » complété par la date courante :

```
public class Ch01_01 {
    public static void main(String[] args) {
        outString = "Bonjour chez vous, le ";
        Calendar rightNow = Calendar.getInstance();
        System.out.println(outString + rightNow.getTime());
    }
}
```

Le lecteur peut identifier d'ores et déjà quelques erreurs. La variable `outString` n'est pas déclarée, ce qui invalide la première et la dernière ligne de la méthode `main` ; de plus, la classe `Calendar` n'a pas été importée, ce qui invalide aussi la deuxième ligne de cette méthode. Un utilisateur faisant appel à `javac` devrait sortir de l'éditeur et exécuter `javac` pour identifier ces erreurs. En revanche, Eclipse identifie celles-ci dès que ces lignes de code sont tapées (voir la figure 1-16).

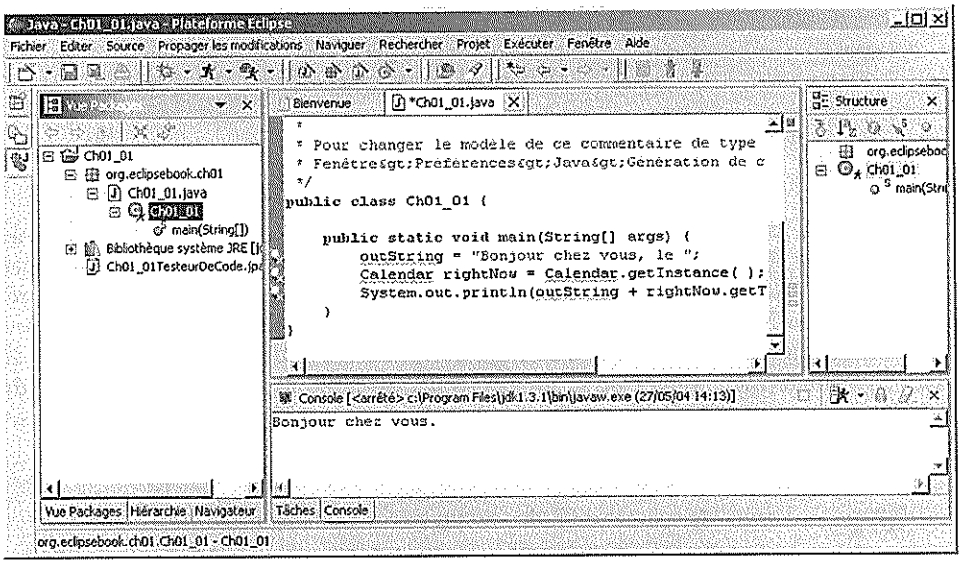


Figure 1-16. Indicateurs Quick Fix

Eclipse ne se contente pas d'indiquer l'existence d'erreurs, mais propose des solutions. Le lecteur remarquera les icônes représentant une petite ampoule jaune et un X rouge, dans la barre verticale se trouvant à gauche de l'éditeur de code : cette barre est appelée *barre de marquage*. Ces icônes indiquent que Quick Fix a localisé des erreurs. Le lecteur remarquera aussi les rectangles rouges évidés se trouvant sur la barre à droite de l'éditeur de code. Ces rectangles indiquent les lignes de code à problème que Quick Fix peut résoudre – les rectangles rouges pleins indiquent les erreurs

de compilation. Le programmeur peut naviguer d'une erreur à l'autre au moyen de ces icônes.

Si l'utilisateur place la souris sur la première icône représentant une petite ampoule dans la barre de marquage, une bulle d'aide apparaît et décrit l'erreur (« outString ne peut pas être résolu »), comme on le voit sur la figure 1-17.

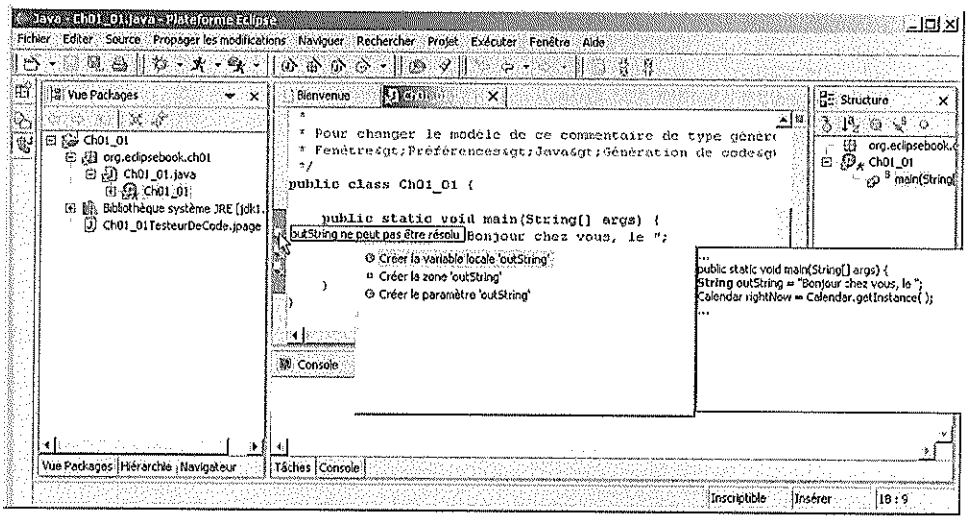


Figure 1-17. Utilisation de Quick Fix

Pour activer Quick Fix et voir les suggestions du JDT, il suffit de cliquer la petite ampoule sur la ligne concernée. Cette action provoque l'affichage d'un certain nombre de suggestions, la première d'entre elles étant « Créer une variable locale 'outString' » (voir la figure 1-17). Quick Fix indique l'effet de la correction proposée dans le contexte du code, comme on peut le voir à droite de la figure. Pour corriger le code, il suffit de double-cliquer sur la proposition « Créer une variable locale 'outString' », ce qui modifie le code dans ce sens, en définissant `outString` comme une variable de type `String` :

```
public class Ch01_01 {
    public static void main(String[] args) {
        String outString = "Bonjour chez vous, le ";
        Calendar rightNow = Calendar.getInstance();
        System.out.println(outString + rightNow.getTime());
    }
}
```

La définition de la variable `outString` résout aussi le problème rencontré dans la dernière ligne de la méthode `main`, qui fait appel à cette variable, si bien que l'ampoule disparaît aussi de cette ligne.

Il reste à résoudre le problème posé par la deuxième ligne de code de la méthode `main`, qui fait appel à la classe `Calendar`. On peut voir ce que Quick Fix suggère dans la figure 1-18 – importer la classe `java.util.Calendar`. Ce seul aspect de Quick Fix justifie pleinement l'apprentissage de cet outil : il devient désormais inutile de partir

à la chasse au paquetage idoine dans toute la hiérarchie proposée par Java, lorsqu'on connaît le nom de la classe utilisée mais pas le paquetage auquel celle-ci appartient.

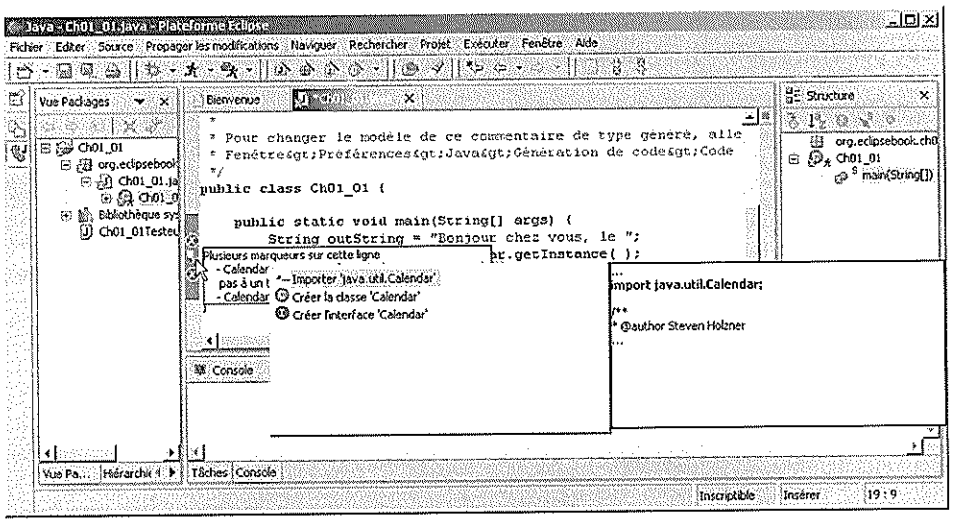


Figure 1-18. Utilisation de Quick Fix pour importer un paquetage

Pour exploiter la suggestion de Quick Fix, il suffit de double-cliquer sur la ligne « Importer 'java.util.Calendar' », ce qui provoque l'importation de la classe `java.util.Calendar` et la modification suivante dans le code :

```
import java.util.Calendar;
        .
        .
        .
public class Ch01_01 {
    public static void main(String[] args) {
        String outString = "Bonjour chez vous, le ";
        Calendar rightNow = Calendar.getInstance();
        System.out.println(outString + rightNow.getTime());
    }
}
```

Après avoir résolu ces problèmes au moyen de Quick Fix, il est possible d'exécuter le code. Afin d'obtenir un affichage conforme aux particularités locales, on peut utiliser la classe `java.text.DateFormat` ; on formate ainsi la date et l'heure grâce aux modifications suivantes :

```
public class Ch01_01 {
    public static void main(String[] args) {
        String outString = "Bonjour chez vous, le ";
        Calendar rightNow = Calendar.getInstance();
       String myString = DateFormat.getDateInstance().format(rightNow.getTime())
    + " à " +
    DateFormat.getTimeInstance().format(rightNow.getTime());
```

```
        System.out.println(outString + myString );
         }
}
```

Comme on le voit dans la figure 1-19, le résultat affiché dans la vue de la console est conforme aux attentes.

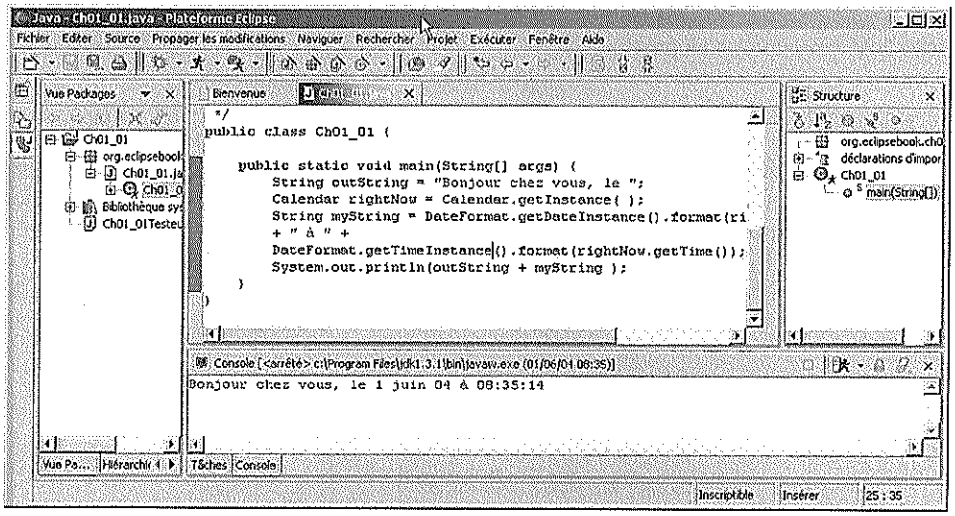


Figure 1-19. exécution du code débogué

Deux mots sur la gestion de projets

Plus le programmeur crée des projets tels que `Ch01_01`, plus l'EDI Eclipse est encombré, car tous les projets sont affichés dans la vue Packages de la perspective Java, et aussi dans la vue Navigateur (qui permet au programmeur de naviguer entre les projets). S'il y a trente projets, il y aura donc trente rubriques différentes. Il existe diverses façons de gérer cette foultitude d'informations (on peut créer par exemple des jeux de documents, comme on peut le voir dans le chapitre 2) ; on va se contenter ici d'examiner la méthode la plus simple.

Pour enlever un projet de la vue Packages ou de la vue Navigateur, il suffit de l'effacer. Cela ne signifie *pas* nécessairement que les fichiers du projet concerné sont réellement effacés ; il est d'ailleurs possible de réintroduire le projet dans ces vues, par la suite. Par exemple, pour enlever le projet `Ch01_01`, il suffit de cliquer le bouton droit de la souris sur l'icône du projet (dans la vue Packages) et de sélectionner le choix `Supprimer`. Eclipse affiche alors une boîte de dialogue demandant confirmation de l'effacement du projet (voir la figure 1-20).

Dans cette situation, il convient de vérifier que le bouton radio « Ne pas supprimer le contenu » est bien sélectionné, avant de cliquer sur le bouton « Oui » pour enlever le projet d'Eclipse. Le projet disparaît alors de la vue Packages et de la vue Navigateur. Cliquer l'autre bouton radio signifie qu'Eclipse doit *effacer* tous les fichiers du projet (et leur contenu), ce qu'il ne faut faire que si l'on ne désire plus *jamais* utiliser le projet et ses fichiers par la suite.

Figure 1-20. Effacement d'un projet

Pour travailler à nouveau avec un projet, il suffit de l'importer. Pour ce faire, il suffit de cliquer le bouton droit de la souris dans la vue Packages ou dans la vue Navigateur et de sélectionner l'option de menu contextuel Importer..., ou de sélectionner l'option de menu Fichier → Importer. Cette action ouvre la boîte de dialogue d'importation ; il suffit alors de sélectionner « Projet existant dans l'espace de travail » et de cliquer Suivant. On clique ensuite le bouton Parcourir, puis l'on sélectionne le dossier *Ch01_01* ; on clique enfin sur OK, ce qui donne le résultat présenté dans la figure 1-21.

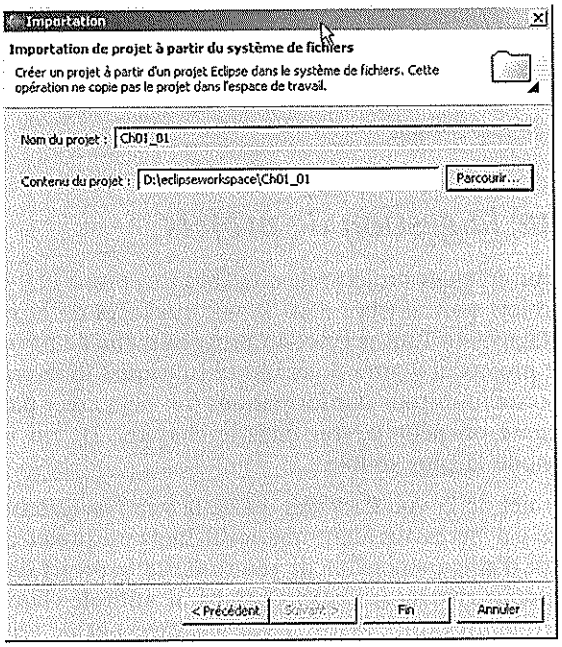


Figure 1-21. Importation d'un projet

On clique alors Fin pour importer à nouveau le projet `Ch01_01`. Celui-ci fait de nouveau partie du monde d'Eclipse et peut être utilisé à tout moment. Pour l'exécuter, on sélectionne le projet dans l'explorateur de paquetages et l'on clique sur le choix de menu Exécuter → Exécuter en tant que → Application Java. Il existe d'autres méthodes pour gérer ses projets ; celles-ci seront examinées plus loin.

On peut aussi fermer un projet en cliquant le bouton droit de la souris sur celui-ci et en sélectionnant le choix de menu Fermer le projet. Cependant, cette action n'enlève pas le projet de la vue Packages ni de la vue Navigateur – le dossier du projet est simplement fermé (une icône représentant un dossier fermé apparaît alors dans la vue Packages). Le seul avantage à fermer un projet est qu'Eclipse se lance plus vite car il n'a pas à initialiser des projets fermés.

Ce chapitre nous a permis de passer en revue les principales caractéristiques d'Eclipse. Le chapitre suivant est consacré au développement d'applications Java et à l'utilisation de JDT.

CHAPITRE 2

Développement d'applications Java

Ce chapitre a pour but d'amener graduellement le lecteur au développement d'applications Java et à la gestion de projet sous Eclipse. Les éléments présentés ici sont pratiquement tous essentiels à une bonne utilisation d'Eclipse.

Développement de code Java

S'il est une chose très consommatrice de temps sous Java, c'est bien l'écriture intégrale d'un code à partir de rien. Bien sûr, la logique d'une méthode, d'une interface ou d'une classe, se traduit par un code unique et développé spécifiquement ; en revanche, les modificateurs d'une méthode, les imports liés à une classe ou la syntaxe introduite par de nouveaux paquetages sont toujours les mêmes. Le programmeur est donc régulièrement amené à retaper les mêmes lignes de code (en y introduisant parfois quelques fautes de frappe), et donc à perdre du temps. Eclipse peut, entre autre, éviter au programmeur cette frappe répétitive.

Création de nouvelles méthodes

Grâce à l'assistant de code, Eclipse facilite la création de nouvelles méthodes. Pour illustrer ce point, nous allons créer et appeler une nouvelle méthode nommée `printer`, qui affiche le message « Bonjour chez vous. », comme on peut le voir dans l'exemple 2-1.

Exemple 2-1. L'exemple Ch02_01.java

```
public class Ch02_01

{
    public static void main(String[] args)
    {
        printer();
    }

    private static void printer()
    {
```

Exemple 2-1. L'exemple Ch02_01.java (suite)

```
        System.out.println("Bonjour chez vous.");
    }
}
```

Comment créer une nouvelle méthode ? Après avoir lancé Eclipse et ouvert un nouveau projet nommé `Ch02_01`, il suffit de créer une nouvelle classe Java, `Ch02_01`, en l'incluant dans le paquetage `org.eclipsebook.ch02` ; la case à cocher indiquant qu'Eclipse doit créer automatiquement une méthode vide (un stub) pour la méthode `main` devra être impérativement cochée. Cela débouche sur le code suivant :

```
public class Ch02_01 {

        public static void main(String[] args) {
        }
}
```

Le programmeur pourrait simplement taper la méthode `printer`, mais Eclipse peut apporter ici son aide. Il suffit de placer le caret sous le code de la méthode `main`, de commencer la frappe avec `private`, ce qui rend la méthode privée, puis de taper Ctrl+Espace pour appeler l'assistant de code. On obtient alors le résultat présenté dans la figure 2-1.

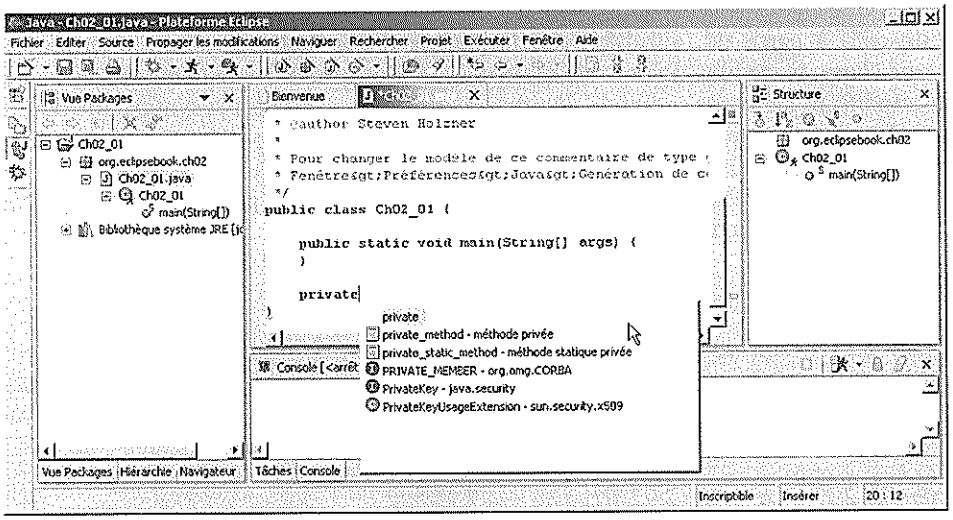


Figure 2-1. Création d'une méthode privée

L'assistant de code permet de sélectionner le type de méthode privée à créer – `private`, `private static`, etc. Si l'on sélectionne ici `private static`, on obtient le modèle de méthode de la figure 2-2. L'identificateur `return_type`, présenté en surbrillance, invite le programmeur à spécifier le type de la valeur retournée : on va choisir ici `void`. Il reste à remplacer l'identificateur `name` par le nom `printer`, puis à effacer la liste d'`arguments`.

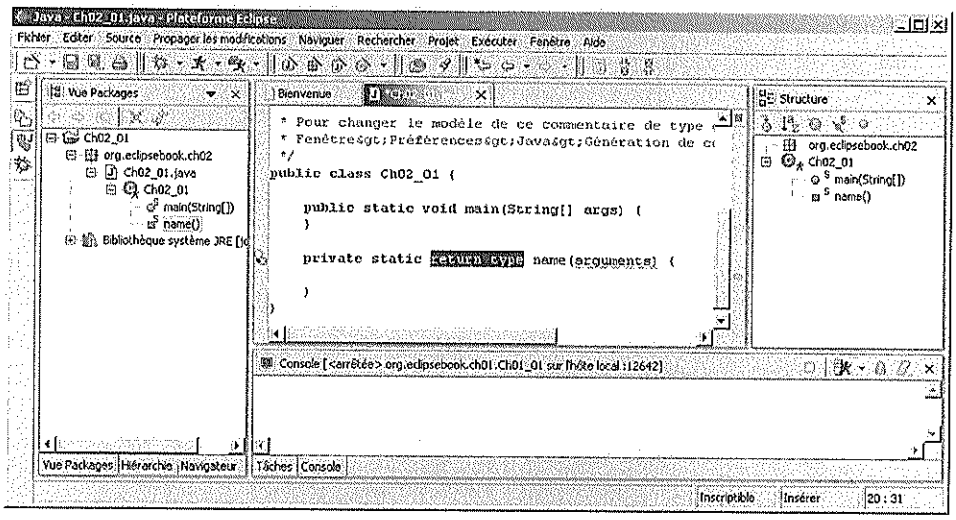


Figure 2-2. Spécification du type retourné par la nouvelle méthode

On termine avec l'ajout du code qui va provoquer l'affichage du message (on peut solliciter l'assistant de code en suspendant un instant la frappe de code après chaque point tapé ; l'assistant formule alors des propositions) :

```
public static void main(String[] args) {
}

private static void printer()
{
    System.out.println("Bonjour chez vous.");
}
```

On peut alors appeler la méthode `printer` dans le `main`:

```
public static void main(String[] args) {
    printer();
}

private static void printer()
{
    System.out.println("Bonjour chez vous.");
}
```

C'est terminé. On peut alors exécuter l'exemple en sélectionnant l'option de menu Exécuter en tant que → Application Java. Le message « Bonjour chez vous. » doit apparaître dans la vue Console – comme précédemment. Cette fois, cependant, on a utilisé une méthode spécifique pour l'affichage.

Création de nouvelles classes

On a créé une nouvelle méthode, on lui a donné un nom, mais comment faire si l'on veut que cette nouvelle méthode soit possédée par une autre classe ? Imaginons, par exemple, que la méthode `main` soit possédée par la classe `Ch02_02`, mais que la

méthode printer soit possédée par une classe nommée Ch02_02Helper :

```
Ch02_02
|
+--------main

Ch02_02Helper
|
+--------printer
```

Dès lors, il est possible de créer une instance de la classe Ch02_02Helper dans la méthode main, et d'appeler la méthode printer de cette instance, comme on peut le voir dans l'exemple 2-2.

Exemple 2-2. L'exemple Ch02_02.java

```
public class Ch02_02 {
        public static void main(String[] args) {
                Ch02_02Helper helper = new Ch02_02Helper();
                helper.printer();
        }
}
```

La classe Ch02_02Helper, qui possède donc la méthode printer, apparaît dans l'exemple 2-3.

Exemple 2-3. L'exemple Ch02_02Helper.java

```
public class Ch02_02Helper {
        public void printer() {
                System.out.println("Bonjour chez vous.");
        }
}
```

Pour implémenter cet exemple basé sur deux classes sous Eclipse, il faut créer un projet nommé Ch02_02 et ajouter la nouvelle classe Ch02_02Helper dans ce projet. Celle-ci, de type public, doit être intégrée dans le paquetage org.eclipsebook.ch02 (il convient de s'assurer qu'aucune méthode main – même vide – n'est créée dans cette classe). Ensuite, la méthode printer est créée dans la classe Ch02_02Helper, comme indiqué dans l'exemple 2-2.

Ensuite, il faut *sauvegarder le fichier*. À ce niveau cette action est très importante, car si elle n'est pas effectuée, la méthode printer ne sera pas reconnue par le reste du code faisant partie de ce projet. Il est essentiel de le savoir : lorsque le programmeur veut utiliser dans un fichier B du code stocké dans un fichier A, il est indispensable de sauvegarder le fichier A au préalable car Eclipse compile le code stocké dans les fichiers et non celui qui est visualisé dans l'éditeur.

Ayant créé la nouvelle classe Ch02_02Helper, il faut maintenant créer la classe Ch02_02, qui utilisera Ch02_02Helper dans ses méthodes. Cette nouvelle classe doit être créée avec une méthode main vide, dans laquelle le programmeur va coder l'instanciation de Ch02_02Helper et l'appel de sa méthode printer :

```
public class Ch02_02 {
        public static void main(String[] args) {
                Ch02_02Helper helper = new Ch02_02Helper();
```

```
            helper.printer();
        }
    }
```

Il faut alors sauvegarder le code ainsi créé et lancer l'exécution de l'application ; le compilateur Eclipse gère lui-même l'appartenance de ces deux classes au même paquetage. On doit voir apparaître le message « Bonjour chez vous. » dans la vue Console (voir figure 2-3). Comme on peut le voir dans la Vue Packages, plusieurs classes ont été utilisées dans le cadre d'un même projet.

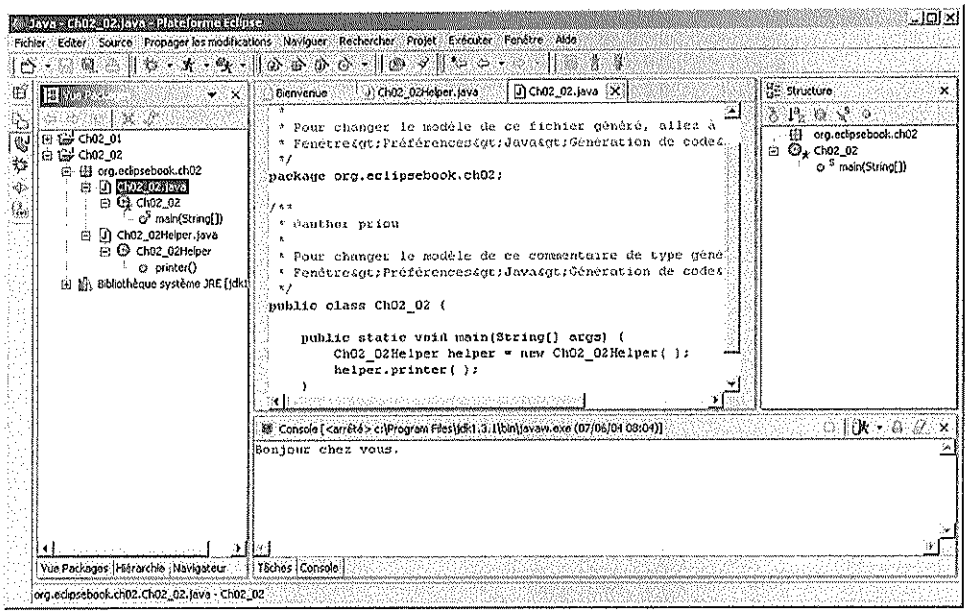


Figure 2-3. Utilisation de plusieurs classes

Et ce n'est pas tout. Imaginons que le programmeur veuille surcharger la méthode `printer` et modifier le texte affiché. Il est possible de dériver la classe de base `Ch02_02Helper` et de surcharger la méthode `printer` dans la classe fille. Pour ce faire, il suffit de cliquer le bouton droit de la souris sur le paquetage `org.eclipsebook.ch02`, dans le projet `Ch02_02`, de sélectionner l'option Nouveau → Classe et d'ouvrir la boîte de dialogue Nouvelle classe Java (voir figure 2-4).

Il faut donner un nom à cette nouvelle classe, par exemple `Ch02_02HelperHelper`, ainsi que le nom du paquetage d'appartenance ; il faut aussi décocher la case déclenchant la création automatique de la méthode `main`. Pour faire en sorte que cette classe dérive de `Ch02_02Helper`, il suffit de taper le nom de la classe mère dans la zone d'édition nommée Superclasse (on peut aussi implémenter des interfaces en tapant leur nom dans la zone de texte nommée Interfaces). On peut alors cliquer sur le bouton Fin : la classe `Ch02_02HelperHelper`, fille de `Ch02_02Helper`, est créée. Voila ce que l'on obtient :

```
public class Ch02_02HelperHelper extends Ch02_02Helper {

}
```

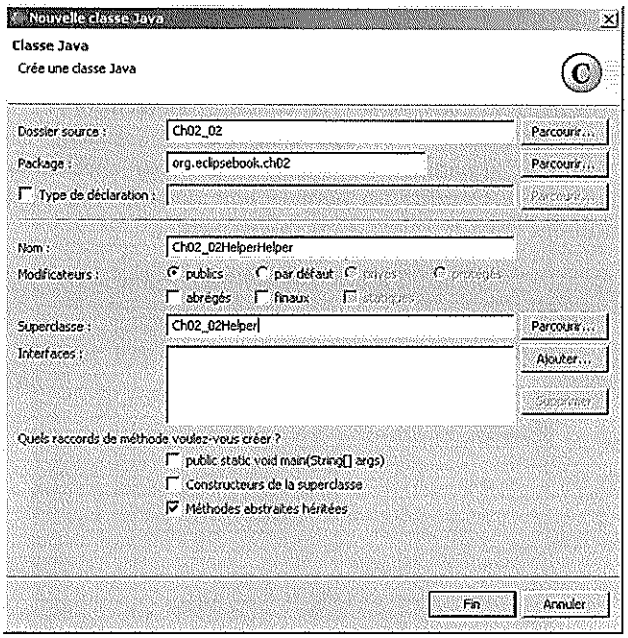


Figure 2-4. Création d'une classe fille

On notera la présence de l'option « Type de déclaration » dans la boîte de dialogue Nouvelle classe Java. Cette option doit être utilisée lorsqu'on veut inclure une classe dans une autre : il faut pour cela taper dans cette zone d'édition le nom de la classe englobante. On peut aussi cliquer le bouton droit de la souris sur la classe englobante, dans la Vue Packages, et sélectionner l'option de menu Nouveau → Classe : le nom de la classe englobante apparaîtra alors dans la zone de texte nommée « Type de déclaration » à l'ouverture de la boîte de dialogue (cette information ne sera utilisée que si la case à cocher ad hoc est cochée).

On va maintenant surcharger la méthode `printer` issue de la classe de base. Pour cela, il faut ouvrir la nouvelle classe, `Ch02_02HelperHelper`, la sélectionner dans la Vue Packages, puis cliquer l'option de menu Source → Remplacer/Implémenter les méthodes : la boîte de dialogue Remplacer/Implémenter les méthodes de la figure 2-5 surgit alors (s'il s'agit d'une interface, le programmeur peut aussi trouver les méthodes à implémenter dans cette boîte de dialogue).

Cette boîte de dialogue affiche la liste des méthodes surchargeables : il suffit d'en sélectionner une. Dans le cas présent, on sélectionne la méthode `printer` puis on clique sur le bouton OK. On voit alors apparaître un stub de la méthode `printer` dans la classe `Ch02_02HelperHelper` :

```
public class Ch02_02HelperHelper extends Ch02_02Helper {

        /* (non-Javadoc)
```

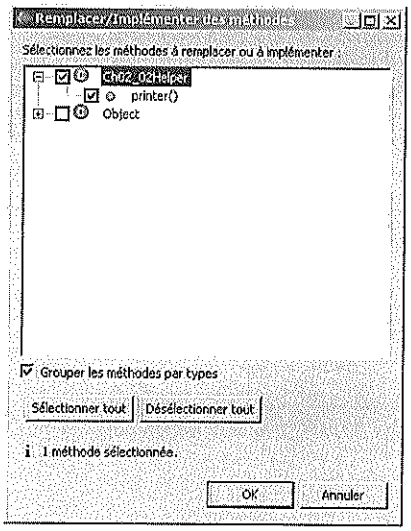


Figure 2-5. Surcharge d'une méthode

```
     * @see org.eclipsebook.ch02.Ch02_02Helper#printer()
     */
    public void printer() {
            // TODO Raccord de méthode auto-généré
            super.printer();
    }

}
```

Dans cette version surchargée, le message affiché dans la vue Console sera « Bonjour tout le monde. »

```
public void printer() {
        System.out.println("Bonjour tout le monde.");
}
```

C'est terminé : on vient de surcharger une méthode dans une classe fille. La dernière étape consiste à utiliser la nouvelle classe dans la méthode `main` :

```
public class Ch02_02 {

        public static void main(String[] args) {
                Ch02_02HelperHelper helper = new Ch02_02HelperHelper();
                helper.printer();
        }
}
```

Comme on peut s'y attendre, l'exécution de la méthode `main` ci-dessus donne le résultat de la figure 2-6.

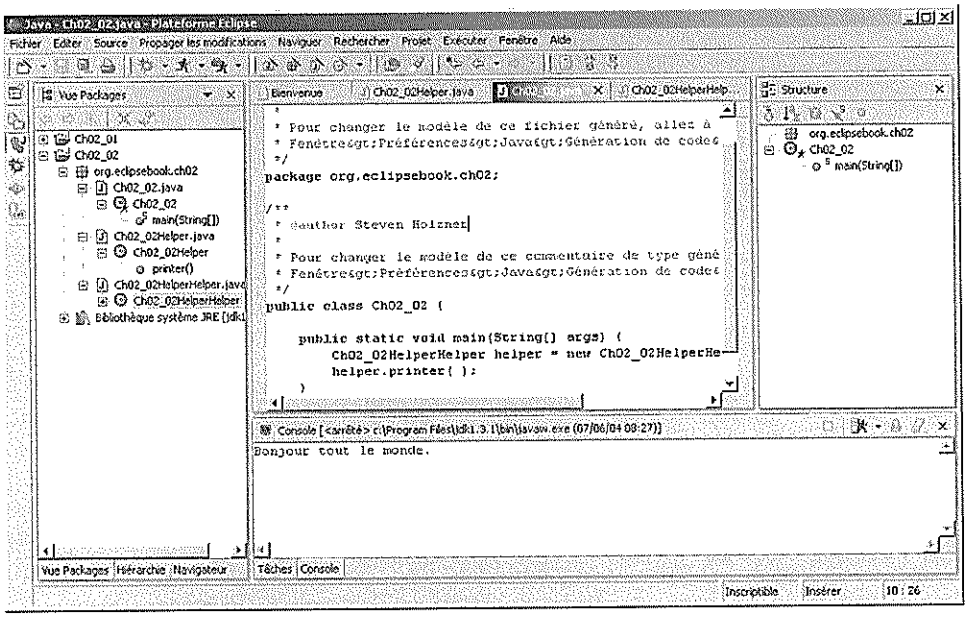


Figure 2-6. Utilisation d'une classe fille

Comme on peut le voir, la génération automatique de code fait gagner du temps au programmeur. Dans le cas présent, on l'a mise en œuvre pour surcharger une méthode, mais le menu Source propose d'autres options :

- L'option Source → Mettre en commentaire permet de commenter une section de code, chose très importante. Il suffit de sélectionner les lignes à commenter, puis de sélectionner cette option de menu. Le symbole C++ du commentaire sur une seule ligne, //, est alors inséré avant chacune de ces lignes de code. Si l'on veut ôter des commentaires, il suffit de sélectionner l'option Source→ Supprimer la mise en commentaire.
- L'option Source → Générer les méthodes Getter et Setter permet de générer les méthodes standard d'accès en lecture ou en écriture à une donnée membre dans une classe (`get...` et `set...`). Par exemple, si la donnée membre concernée s'appelle `temperature`, ces méthodes s'appelleront probablement `getTemperature` et `setTemperature`.
- L'option Source → Générer des méthodes déléguées permet de créer automatiquement des méthodes déléguées pour d'autres méthodes.
- L'option Source → Ajouter un constructeur de la superclasse permet d'inclure l'appel du constructeur de la classe parent (on peut aussi créer ce constructeur automatiquement à la création de la classe).
- L'option Source → Entourer d'un bloc try/catch est véritablement remarquable : celle-ci permet d'intégrer du code dans une zone de try/catch. Cette option vérifie automatiquement les exceptions non traitées et génère la zone de catch idoine.

Imaginons par exemple que le message affiché par la méthode `printer` soit stocké dans une donnée membre nommée `message`:

```
public class Ch02_02HelperHelper extends Ch02_02Helper {

        /* (non-Javadoc)
         * @see org.eclipsebook.ch02.Ch02_02Helper#printer()
         */

    String message = "Bonjour tout le monde.";

        public void printer() {
                System.out.println(message);
        }
```

Au lieu d'accéder directement à cette donnée membre, on peut créer et utiliser les méthodes `getMessage` et `setMessage` (ces méthodes ont la forme standard JavaBeans™). Pour cela, il suffit de sélectionner l'option de menu Source → Générer les méthodes Getter et Setter, qui ouvre la boîte de dialogue du même nom (voir figure 2-7), de cocher la case se trouvant face à `message` et de cliquer sur le bouton OK.

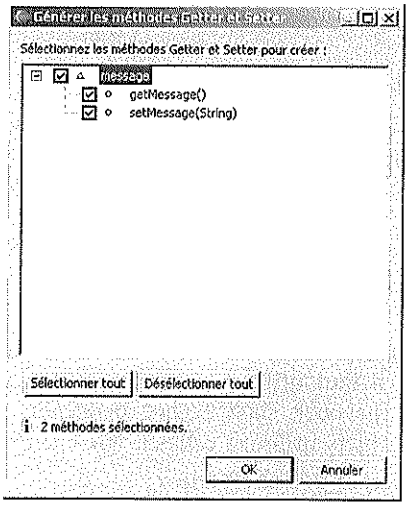


Figure 2-7. Génération des méthodes Getter et Setter

Eclipse crée alors les méthodes `getMessage` et `setMessage` ; on peut utiliser cette dernière dans le code de la méthode `printer` :

```
public class Ch02_02HelperHelper extends Ch02_02Helper {

        /* (non-Javadoc)
         * @see org.eclipsebook.ch02.Ch02_02Helper#printer()
         */

        String message = "Bonjour tout le monde.";
```

```
public void printer() {
        System.out.println(getMessage());
}

/**
 * @return
 */
public String getMessage() {
        return message;
}

/**
 * @param string
 */
public void setMessage(String string) {
        message = string;
}
```

L'option de menu Source → Entourer d'un bloc try/catch permet aussi au programmeur de gagner beaucoup de temps. Cette option vérifie automatiquement, dans une zone de code sélectionnée par le programmeur, les exceptions non capturées et génère un bloc try/catch pour chacune d'entre elles. Cet outil est véritablement remarquable et justifie pleinement d'investir du temps dans l'apprentissage d'Eclipse.

Création de nouveaux paquetages

Au-delà de la création de plusieurs classes dans un même projet, il est aussi très facile, dans le cadre d'un même projet, de stocker du code dans plusieurs *paquetages*. Par exemple, on peut scinder le code de notre exemple en deux paquetages, `org.eclipsebook.ch02` et `org.eclipsebook.ch02_2`, comme on peut l'observer dans la Vue Packages, sur la partie gauche de la figure 2-8. Ici, la classe `Ch02_03Helper` et sa méthode `printer` sont localisées dans un paquetage entièrement nouveau, nommé `org.eclipse.ch02_2`.

Si l'on veut accéder à une classe appartenant à un autre paquetage, il faut importer celui-ci – dans l'exemple, il faut utiliser le nom complet, paquetage compris, de la classe, c'est-à-dire `org.eclipsebook.ch02_2.Ch02_03Helper`. Après avoir importé cette classe, on peut créer des objets utilisant celle-ci. Par exemple :

```
import org.eclipsebook.ch02_2.Ch02_03Helper;

/**
 * @author Steven Holzner
 *
 * Pour changer le modèle de ce commentaire de type généré, allez à :
 * Fenêtre>Préférences>Java>Génération de code>Code et commentaires
 */
public class Ch02_03 {

        public static void main(String[] args){
                Ch02_03Helper helper = new Ch02_03Helper();
                helper.printer();
        }
}
```

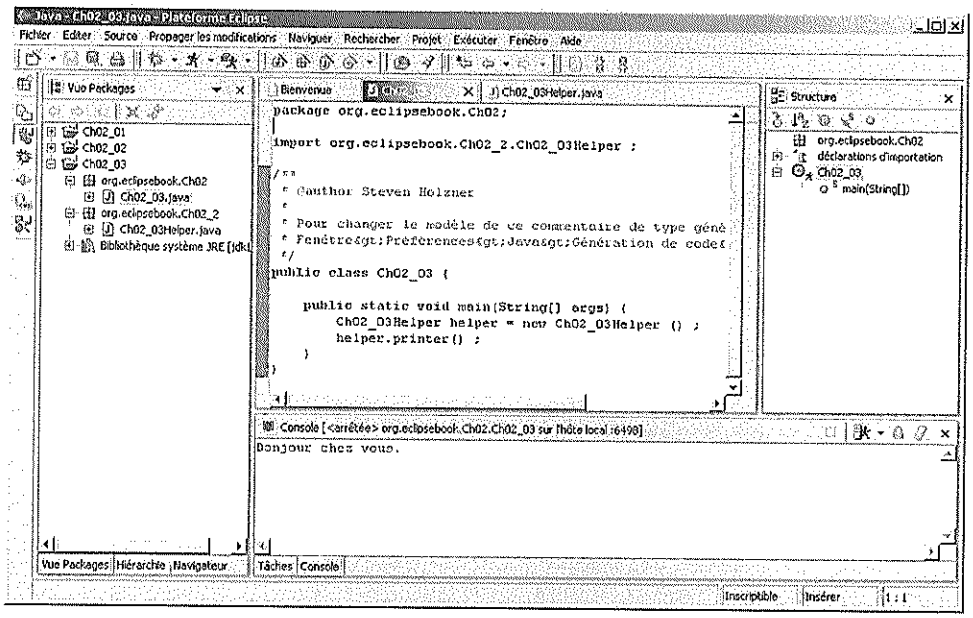


Figure 2-8. Utilisation de plusieurs paquetages

C'est tout. L'utilisation de plusieurs paquetages dans le même projet ne pose aucun problème.

Que faire lorsque le code à utiliser est non seulement dans un autre paquetage, mais aussi dans un autre projet ? Lors de la création d'un projet, une option permet d'ajouter d'autres projets au chemin de compilation. Par exemple, si l'on veut créer un projet, nommé `Ch02_03a`, qui utilise et modifie le code du paquetage `org.eclipsebook.ch02_2`, il suffit de cliquer l'onglet Projets dans le deuxième panneau de la boîte de dialogue Nouveau projet Java et d'ajouter le projet `Ch02_03` dans le chemin de compilation, comme on peut le voir dans la figure 2-9.

A présent, le code de `Ch02_03` est accessible pour le nouveau projet (y compris le paquetage `org.eclipsebook.ch02_2`). On peut ajouter aussi d'autres paquetages dans le chemin de compilation après la création d'un projet. Pour cela, il suffit de sélectionner le projet concerné dans la Vue Packages, de cliquer le bouton droit de la souris dessus et de sélectionner l'option de menu contextuel Propriétés : la boîte de dialogue du même nom s'ouvre. On sélectionne ensuite Chemin de compilation Java, on clique l'onglet Projets, ce qui donne le résultat présenté dans la figure 2-9.

Compilation et exécution du code

Comment créer les fichiers *.class* à partir des fichiers *.java* d'un projet ? Il suffit d'employer les options du menu Projet. La principale d'entre elles est Projet → Recompiler un projet, qui compile les fichiers du code source du projet, génère les fichiers *.class* et les range dans le même répertoire que celui des fichiers source par défaut. Par exemple, si l'on travaille sur le projet `Ch02_03` et si l'on sélectionne l'option Projet→ Recompiler un projet, les fichiers *.class* associés à ce projet seront

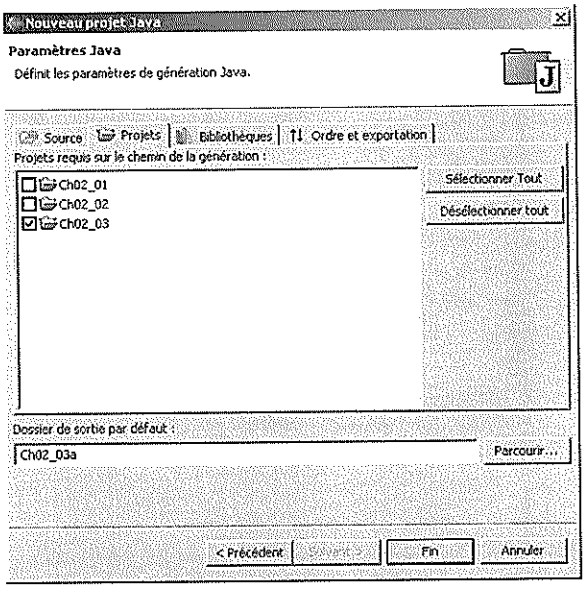


Figure 2-9. Ajout d'un projet dans le chemin de compilation

créés dans le répertoire *workspace/Ch02_03/org/eclipsebook/Ch02* (on n'oubliera pas que les classes de ce projet sont stockées dans le paquetage `org.eclipsebook.Ch02`, d'où la présence de */Ch02* dans le chemin d'accès). Une fois créés, les fichiers *.class* peuvent être diffusés.

Il est souvent justifié de stocker tous les fichiers de code associés à un projet dans un répertoire nommé *src* et tous les fichiers de code binaire engendrés dans un répertoire nommé *bin*. Pour faire en sorte que cela soit bien le cas, il faut, lors de la création d'un projet, effectuer quelques choix spécifiques dans le deuxième panneau de la boîte de dialogue Nouveau projet Java. Ayant ouvert celle-ci, il faut cliquer l'onglet Source et ensuite le bouton Ajouter un dossier : il apparaît une boîte de dialogue nommée Sélection du dossier source, dans laquelle on doit cliquer sur le bouton Créer un dossier et taper *src* comme nom du nouveau dossier. On clique enfin OK deux fois : Eclipse va comprendre que le programmeur veut créer un nouveau répertoire destiné à recevoir les fichiers de code source ; le programmeur se verra même proposer de créer un répertoire *bin* destiné à recevoir les fichiers binaires engendrés, comme on peut le voir dans la figure 2-10.

Configurer ainsi un projet fait que tous les fichiers source seront *automatiquement* stockés dans le répertoire *src* et que tous les fichiers binaires engendrés le seront dans le répertoire *bin* (le répertoire *bin* n'apparaîtra pas dans la Vue Packages car il ne contient pas de code source).

Utilisation de fichiers JAR et .class

Imaginons devoir développer le servlet Java présenté dans l'exemple 2-4 (pour plus de détails sur les servlets, voir chapitre 9).

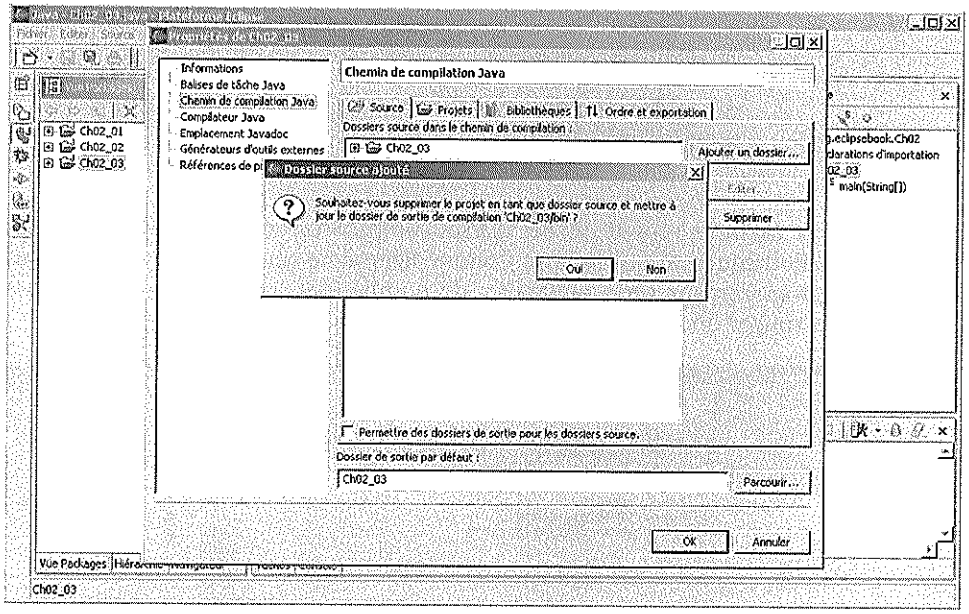


Figure 2-10. Création des répertoires source et bin

Exemple 2-4. L'exemple Ch02_04.java

```
package org.eclipsebook.Ch02;

import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

/**
 * @author Steven Holzner
 *
 * Pour changer le modèle de ce commentaire de type généré, allez à :
 * Fenêtre>Préférences>Java>Génération de code>Code et commentaires
 */

public class Ch02_04 extends HttpServlet {

        public void doGet(HttpServletRequest request,
                HttpServletResponse response)
                throws IOException, ServletException
        {
                response.setContentType("text/html");
                PrintWriter out = response.getWriter();

                out.println("<HTML>");
                out.println("<HEAD>");
                out.println("<TITLE>");
                out.println("A Web Page");
                out.println("</TITLE>");
```

Exemple 2-4. L'exemple Ch02_04.java (suite)

```
                out.println("</HEAD>");
                out.println("Hello there!");
                out.println("</BODY>");
                out.println("</HTML>");
            }
    }
```

Le code de cet exemple repose sur des classes comme `HttpServlet`, qui ne font pas partie des bibliothèques du noyau Java. Comme Eclipse ne peut pas trouver les classes requises, le programmeur verra apparaître de nombreuses lignes soulignées par des petites lignes rouges ondulées (voir figure 2-11).

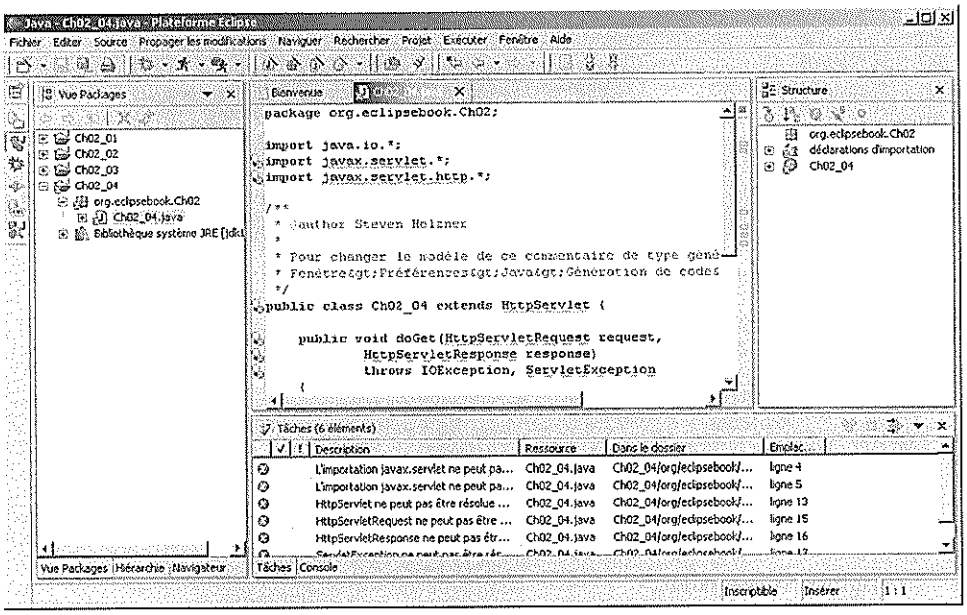


Figure 2-11. Présence de nombreuses erreurs de type "L'importation ... ne peut être résolue"

On peut résoudre ce problème très facilement en incluant le fichier JAR (Java Archive) idoine dans le classpath ; dans le cas présent, le fichier recherché est *servlet.jar*. Ce fichier JAR est fourni par Tomcat, le serveur d'applications qui sera utilisé plus loin dans cet ouvrage. Pour ajouter *servlet.jar* au classpath, il suffit de cliquer le bouton droit de la souris sur le projet, dans la Vue Packages, et de sélectionner l'option Propriétés, ce qui ouvre la boîte de dialogue présentée dans la figure 2-12.

Dans le cas présent, il faut sélectionner l'onglet Bibliothèques de cette boîte de dialogue, cliquer Ajouter des fichiers JAR externes, naviguer et trouver *servlet.jar* et enfin cliquer le bouton OK. Le fichier *servlet.jar* est alors ajouté dans le classpath, comme on peut le voir dans la figure 2-12. On clique alors OK pour fermer la boîte de dialogue Propriétés et on compile le projet : le résultat (correct) de toutes ces opérations est présenté dans la figure 2-13, car le fichier JAR requis est maintenant dans le classpath – on peut noter la référence à *servlet.jar* dans ce projet au moyen de la Vue Packages.

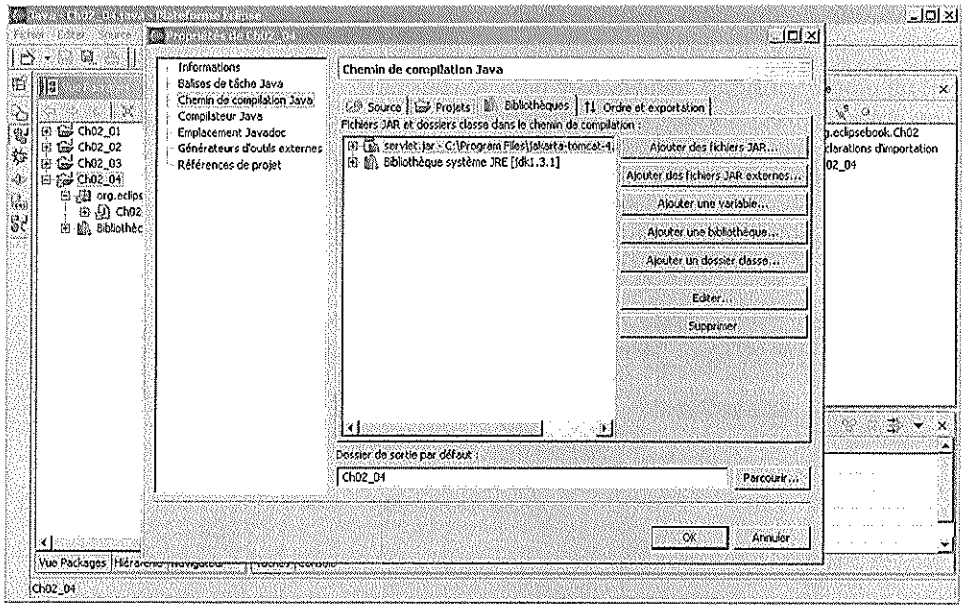


Figure 2-12. la boîte de dialogue Propriétés

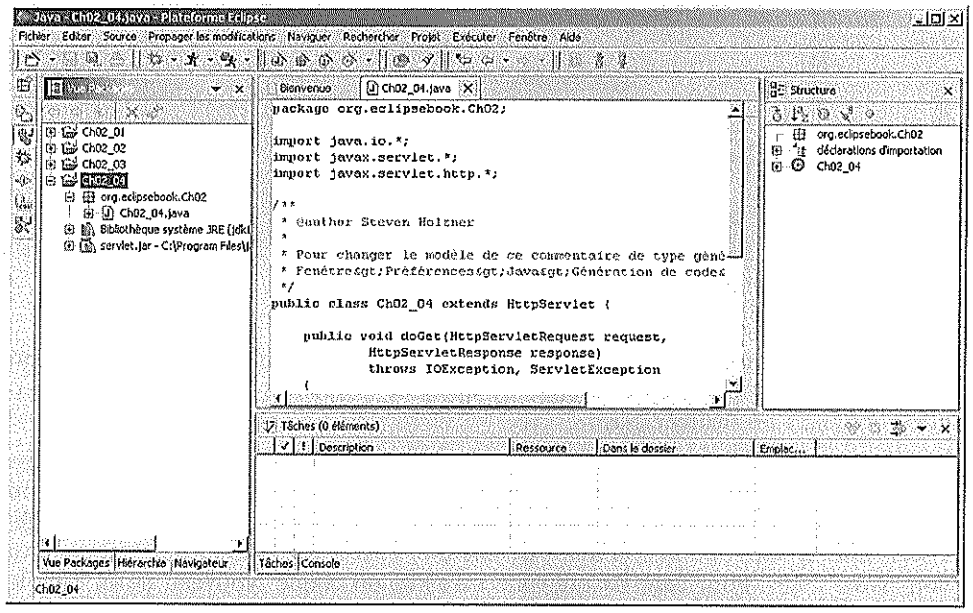


Figure 2-13. Un nouveau fichier JAR file dans le chemin de compilation

Si dès la création d'un projet, le programmeur sait qu'il devra utiliser un fichier JAR tel que *servlet.jar*, il peut ajouter ce fichier JAR dans le classpath du projet au moyen du troisième panneau de la boîte de dialogue Nouveau Projet. On y retrouve les

mêmes onglets que dans la figure 2-12 – il suffit de cliquer l'onglet Bibliothèques pour ajouter tout fichier JAR requis par le projet.

Si l'on ajoute plusieurs fichiers JAR dans le classpath, on peut aussi indiquer l'ordre dans lequel ceux-ci doivent être recherchés – il suffit de sélectionner l'onglet Ordre et exportation dans la boîte de dialogue Propriétés (voir figure 2-14) et de changer l'ordre des composants importés au moyen des boutons Haut et Bas. On peut aussi indiquer les composants à exporter au moyen des cases à cocher de cette même boîte ; lorsqu'un composant est exporté, il devient disponible pour tout projet dépendant du projet courant.

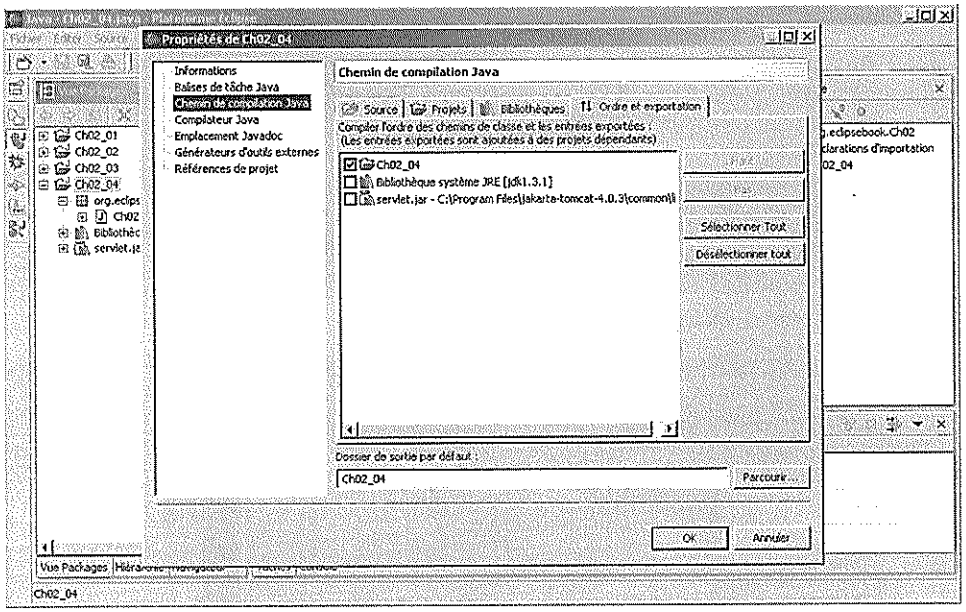


Figure 2-14. Etablissement de l'ordre d'importation

On peut aussi ajouter des fichiers *.class* au classpath – il suffit d'utiliser le bouton Ajouter un dossier classe dans la boîte de dialogue Propriétés pour ajouter le répertoire contenant les fichiers *.class*.

Imaginons devoir développer plusieurs servlets ; il peut s'avérer pratique de créer une *variable de classpath* correspondant à *servlet.jar* et permettant d'ajouter ce fichier au classpath. Pour créer une variable de classpath, il faut sélectionner Fenêtre → Préférences, puis la sous-option Variables de chemin d'accès aux classes de l'option Java (voir figure 2-15). Pour créer la nouvelle variable, on clique Nouveau et on tape le nom de celle-ci – dans notre exemple, la variable s'appelle `SERVLET_LIB` – ainsi que le chemin correspondant, puis on clique le bouton OK. On obtient le résultat de la figure 2-15.

À présent, pour ajouter la variable de classpath au classpath d'un projet, il suffit d'ouvrir la boîte de dialogue Propriétés, de cliquer l'onglet Bibliothèques, puis le bouton Ajouter une variable, que l'on peut voir dans la figure 2-12, puis de sélectionner la variable de classpath concernée. L'utilisation de telles variables de classpath

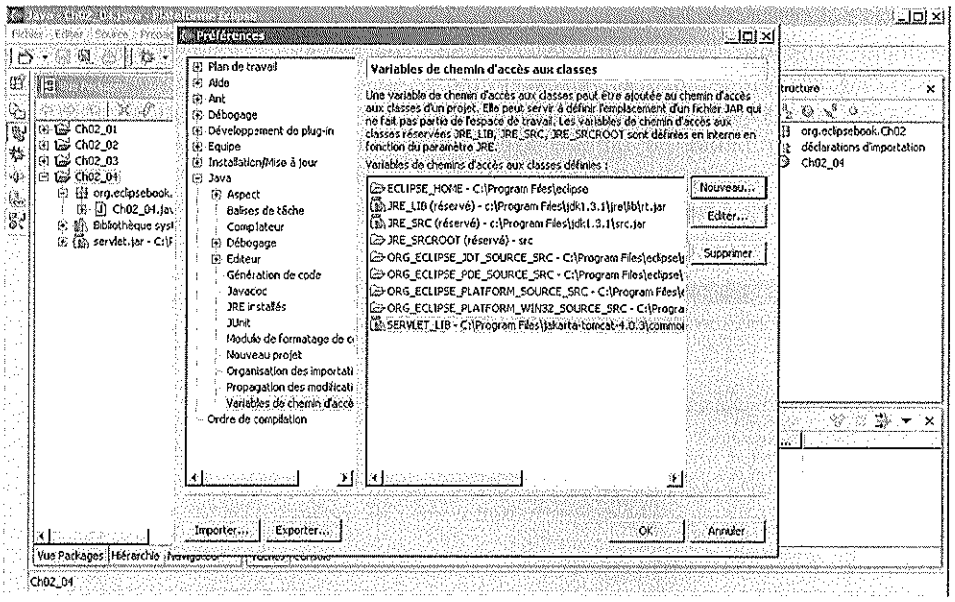


Figure 2-15. Création d'une variable de classpath

est très pratique car elle permet de centraliser toutes les références de classpath – par exemple, si l'on veut utiliser une nouvelle version de *servlet.jar*, il suffit de mettre à jour la seule variable de classpath associée.

Définition de la configuration d'exécution

Imaginons que le code à développer nécessite la lecture d'arguments en ligne de commande (voir exemple 2-5). Cet exemple affiche le premier lexème qui lui est passé en ligne de commande. On peut lancer ce programme en ligne de commande de la façon suivante : `%java Ch02_05 Bonjour!` (dans cet ouvrage, le symbole % sera utilisé en guise de prompt de ligne de commande). Le programme doit alors afficher le lexème « Bonjour! ».

Exemple 2-5. L'exemple Ch02_05.java

```
package org.eclipsebook.ch02;

/**
 * @author Steven Holzner
 *
 * Pour changer le modèle de ce commentaire de type généré, allez à :
 * Fenêtre>Préférences>Java>Génération de code>Code et commentaires
 */
public class Ch02_05 {

        public static void main(String[] args) {
                System.out.println(args[0]);
        }
}
```

Exemple 2-5. L'exemple Ch02_05.java (suite)

Malheureusement, l'exécution de ce code déclenche l'affichage de l'erreur `java.lang.ArrayIndexOutOfBoundsException at org.eclipsebook.ch02.Ch02_05.main(Ch02_05.java:18)` dans la vue Console, car on n'a fourni aucun argument en ligne de commande au tableau args. Pour remplir celui-ci, il faut modifier la configuration d'exécution du projet ; pour cela, il suffit de sélectionner l'option de menu Exécuter → Exécuter..., qui ouvre la boîte de dialogue Exécuter de la figure 2-16.

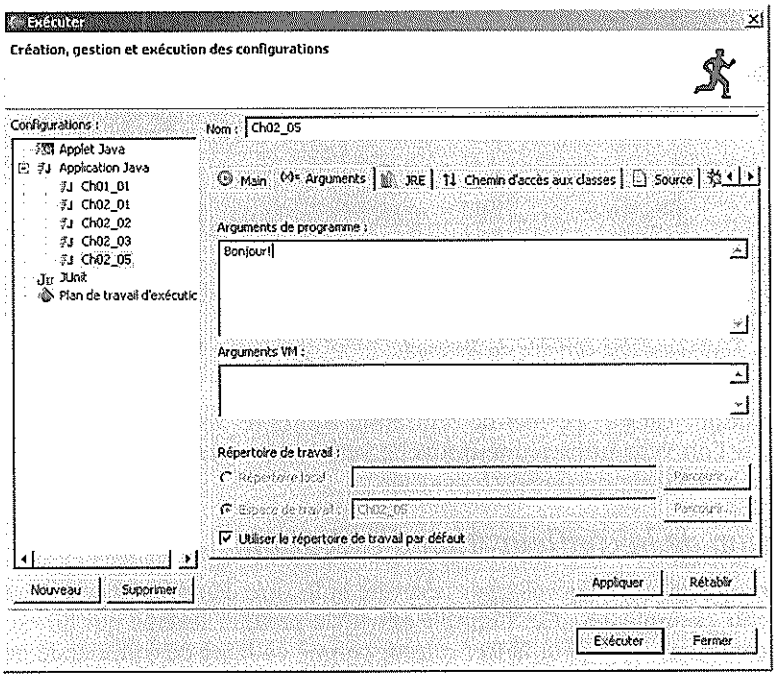


Figure 2-16. Définition de la configuration d'exécution

On doit alors taper le message « Bonjour! » dans cette boîte de dialogue (voir figure 2-16). On peut aussi saisir les options de la Machine Virtuelle Java dans la zone de texte intitulée Arguments VM, ou encore sélectionner l'environnement de JRE Java de son choix en cliquant l'onglet JRE. Après avoir tapé « Bonjour! » dans la zone de texte intitulée Arguments de programme, on clique Exécuter : le programme peut alors lire l'argument en ligne de commande et l'afficher (voir figure 2-17).

Sélection du JRE Java

Lorsque Eclipse est lancé pour la première fois, Eclipse cherche les JRE Java installés sur l'ordinateur : il peut très bien en utiliser un qui ne convient pas au programmeur. Par exemple, Eclipse peut sélectionner une ancienne version du JRE, fournie avec l'explorateur de l'ordinateur, au lieu du SDK Java tout récent que le programmeur vient de télécharger. On peut spécifier le JRE Java que doit utiliser Eclipse en sélectionnant Fenêtre → Préférences, puis JRE installés, et enfin le JRE concerné, comme

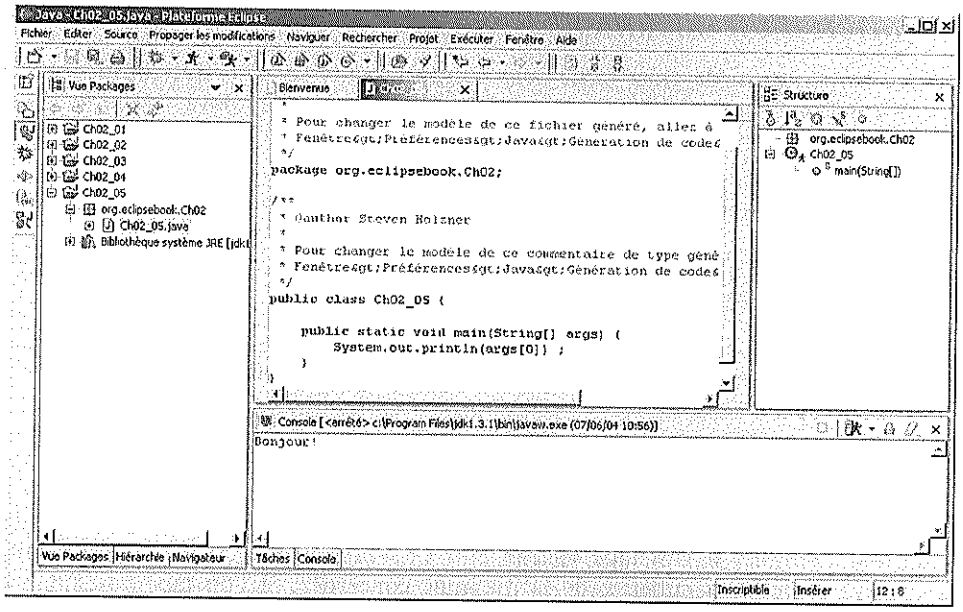


Figure 2-17. Fourniture d'arguments en ligne de commande à un programme

indiqué dans la figure 2-18 (utiliser le bouton Ajouter pour ajouter des JRE et des SDK).

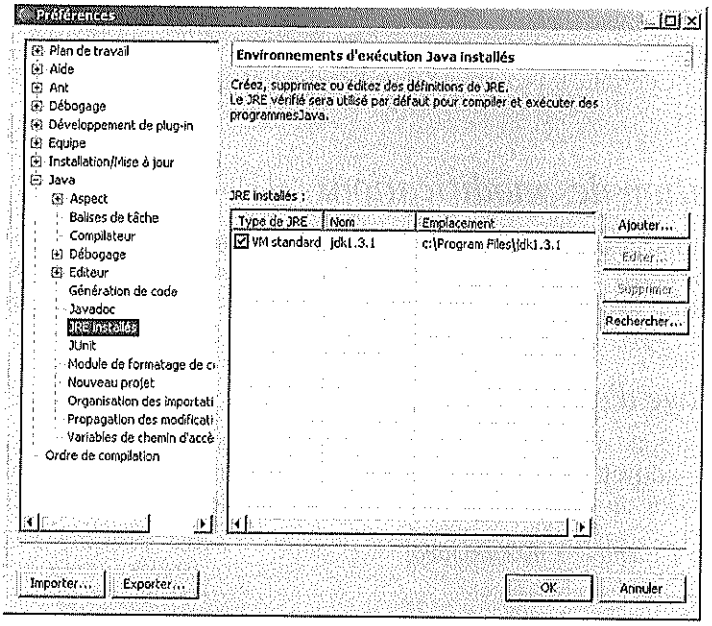


Figure 2-18. Sélection d'un JRE

Création d'un document Javadoc

Eclipse permet aussi de créer facilement des documents Javadoc, qui constituent la documentation standard d'accompagnement des programmes Java. Dans le code généré par Eclipe, on notera la présence de texte Javadoc, comme on peut le voir dans *Ch02_05.java* :

```
package org.eclipsebook.ch02;

/**
 * @author Steven Holzner
 *
 * Pour changer le modèle de ce commentaire de type généré, allez à :
 * Fenêtre>Préférences>Java>Génération de code>Code et commentaires
 */
.
.
.
```

Pour taper son propre texte Javadoc, le programmeur peut s'aider de l'assistant de code. Par exemple, si l'on tape `@param` puis Ctrl+Espace pour invoquer l'assistant, celui-ci va lister les paramètres de la méthode concernée. Si l'on tape `@exception`, l'assistant de code va lister les exceptions levées par la méthode, etc. Si l'on tape `@` dans un commentaire, et si l'on suspend la frappe un instant, l'assistant de code va lister les mots-clés Javadoc, comme `@author`, `@deprecated`, etc.

Pour générer des commentaires Javadoc à partir d'un fichier de code, il faut sélectionner l'option Projet → Générer le Javadoc..., qui ouvre la boîte de dialogue Génération d'un Javadoc ; celle-ci permet de sélectionner le projet à partir duquel on veut générer un document Javadoc. Pour explorer les documents Javadoc associés à un projet, on sélectionne l'option de menu Naviguer → Ouvrir le Javadoc externe. On peut voir, par exemple, les commentaires Javadoc générés pour le projet `Ch02_05` dans la figure 2-19.

Propagation des modifications

L'un des avantages majeurs d'un bon EDI tel qu'Eclipse est qu'il permet au programmeur de renommer et de déplacer du code Java, les conséquences de ces modifications (renommage ou déplacement) sur l'ensemble du code étant gérées automatiquement par l'EDI.

Renommage d'éléments

Examinons le code de l'exemple 2-6, pour lequel l'assistant de code a été sollicité afin de créer une nouvelle méthode chargée d'afficher un message simple. Imaginons que le programmeur ait oublié de modifier le nom par défaut de cette méthode.

Exemple 2-6. L'exemple Ch02_06.java

```
package org.eclipse.Ch02;

/**
 * @author Steven Holzner
```

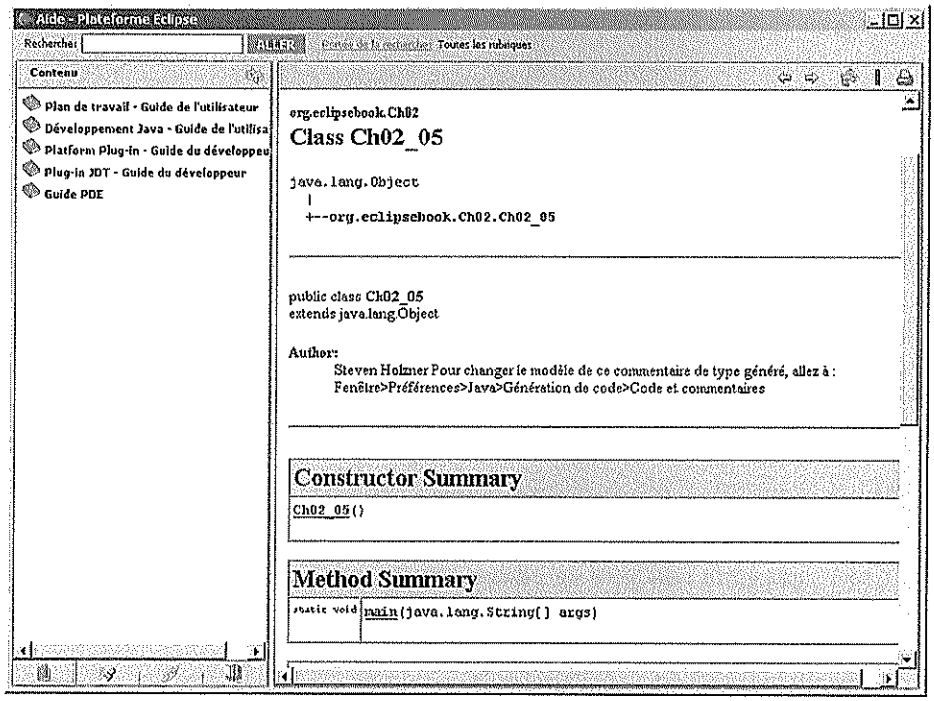


Figure 2-19. Navigation dans un document Javadoc

Exemple 2-6. L'exemple Ch02_06.java (suite)

```
    *
    * Pour changer le modèle de ce commentaire de type généré, allez à :
    * Fenêtre>Préférences>Java>Génération de code>Code et commentaires
    */
public class Ch0206 {

        public static void main(String[] args) {
                name();
        }

        public static void name() {
                System.out.println("Bonjour chez vous.");
        }
}
```

Ce nom par défaut (name) est employé dans la méthode main ; il pourrait l'être en tout autre endroit du code. Comment renommer la méthode name et mettre à jour tous les appels de celle-ci dans le code ? Il suffit pour cela de sélectionner name dans l'éditeur, puis l'option de menu Propager les modifications → Renommer... : ceci ouvre la boîte de dialogue Changement de nom de la méthode présentée dans la figure 2-20.

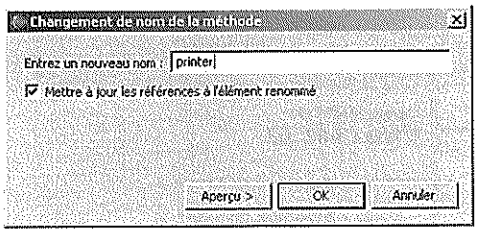


Figure 2-20. Propagation des modifications pour une méthode

Il suffit alors de taper le nouveau nom de la méthode, `printer` dans le cas présent, et de cliquer le bouton OK. Le nom de la méthode est alors modifié, ainsi que tous les appels de celle-ci dans le code du projet, comme on peut le voir ici :

```
package org.eclipse.ch02;

/**
 * @author Steven Holzner
 *
 * Pour changer le modèle de ce commentaire de type généré, allez à :
 * Fenêtre>Préférences>Java>Génération de code>Code et commentaires
 */
public class Ch0206 {

        public static void main(String[] args) {
                printer();
        }

        public static void printer() {
                System.out.println("Bonjour chez vous.");
        }
}
```

Dans cet exemple, la classe a été mal nommée – `Ch0206`, au lieu de `Ch02_06`. Pour renommer celle-ci, il suffit de sélectionner `Ch0206` dans l'éditeur, puis l'option de menu Propager les modifications → Renommer ; la boîte de dialogue Changement du nom du type de la figure 2-21 apparaît alors. On tape le nouveau nom de la classe, `Ch02_06`, on clique sur le bouton OK et le tour est joué.

Cliquer sur OK change non seulement le nom de la classe dans le code, mais aussi le nom du fichier de la classe (*Ch0206.java* devient *Ch02_06.java*), comme on peut le vérifier dans la Vue Packages. Voici la nouvelle version du code :

```
package org.eclipse.ch02;

/**
 * @author Steven Holzner
 *
 * Pour changer le modèle de ce commentaire de type généré, allez à :
 * Fenêtre>Préférences>Java>Génération de code>Code et commentaires
 */
public class Ch02_06 {
```

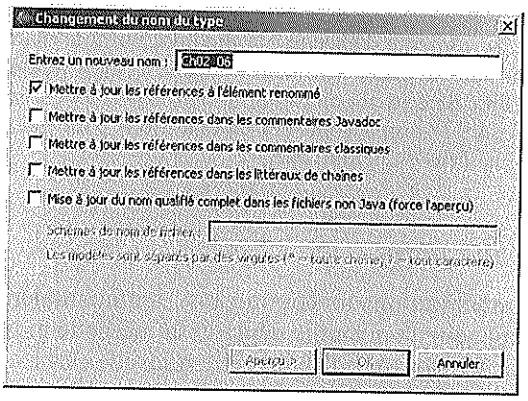


Figure 2-21. Propagation des modifications pour une classe

```
        public static void main(String[] args) {
                printer();
        }

        public static void printer() {
                System.out.println("Bonjour chez vous.");
        }
}
```

En fait, on a aussi malencontreusement mal nommé le paquetage, lors de la création de ce code d'exemple – `org.eclipse.ch02` au lieu de `org.eclipsebook.ch02`. En utilisant la propagation de modifications sur celui-ci, le nom de ce paquetage va être modifié dans la Vue Packages et partout dans le code :

```
package org.eclipsebook.ch02;

/**
 * @author Steven Holzner
 *
 * Pour changer le modèle de ce commentaire de type généré, allez à :
 * Fenêtre>Préférences>Java>Génération de code>Code et commentaires
 */
public class Ch02_06 {

        public static void main(String[] args) {
                printer();
        }

        public static void printer() {
                System.out.println("Bonjour chez vous.");
        }
}
```

Comme on peut le voir, il est très facile de renommer les éléments d'un code Java – Eclipse gère automatiquement tous les détails, en particulier les conséquences du renommage sur tout le code.

Si l'on se contente d'effectuer la modification à la main directement dans le code, il n'y aura aucune gestion automatique par Eclipse de cette modification. Il faut explicitement solliciter la fonction de propagation des modifications pour que celle-ci agisse sur l'ensemble du code.

Déplacement d'éléments

La propagation des modifications fonctionne aussi automatiquement sur les fichiers. Imaginons par exemple que l'on veuille déplacer la méthode `printer` dans une autre classe, `Ch02_06Helper`. Créons tout d'abord cette nouvelle classe, qu'Eclipse va stocker dans un fichier propre, *Ch02_06Helper.java*. On sélectionne la méthode à déplacer, `printer`, en sélectionnant son nom, « printer », dans son prototype. On sélectionne ensuite Propager les modifications→ Déplacer... pour ouvrir la boîte de dialogue de la figure 2-22. Pour déplacer la méthode vers la classe `Ch02_06Helper`, il faut taper le nom complet de celle-ci, `org.eclipsebook.ch02.Ch02_06Helper`, et cliquer sur le bouton OK. La méthode `printer` est alors déplacée vers la classe `Ch02_06Helper` de la façon suivante :

```
package org.eclipsebook.ch02;

/**
 * @author Steven Holzner
 *
 * Pour changer le modèle de ce commentaire de type généré, allez à :
 * Fenêtre>Préférences>Java>Génération de code>Code et commentaires
 */
public class Ch02_06Helper {

        public static void printer() {
                System.out.println("Bonjour chez vous.");
        }
}
```

Agir de la sorte fait que tous les appels de la méthode `printer` sont correctement mis à jour, au sens où c'est bien `Ch02_06Helper.printer` qui est appelée, après coup, dans la méthode `main` de la classe `Ch02_06` :

```
package org.eclipsebook.ch02;

/**
 * @author Steven Holzner
 *
 * Pour changer le modèle de ce commentaire de type généré, allez à :
 * Fenêtre>Préférences>Java>Génération de code>Code et commentaires
 */
public class Ch02_06 {

        public static void main(String[] args) {
                Ch02_06Helper.printer();
```

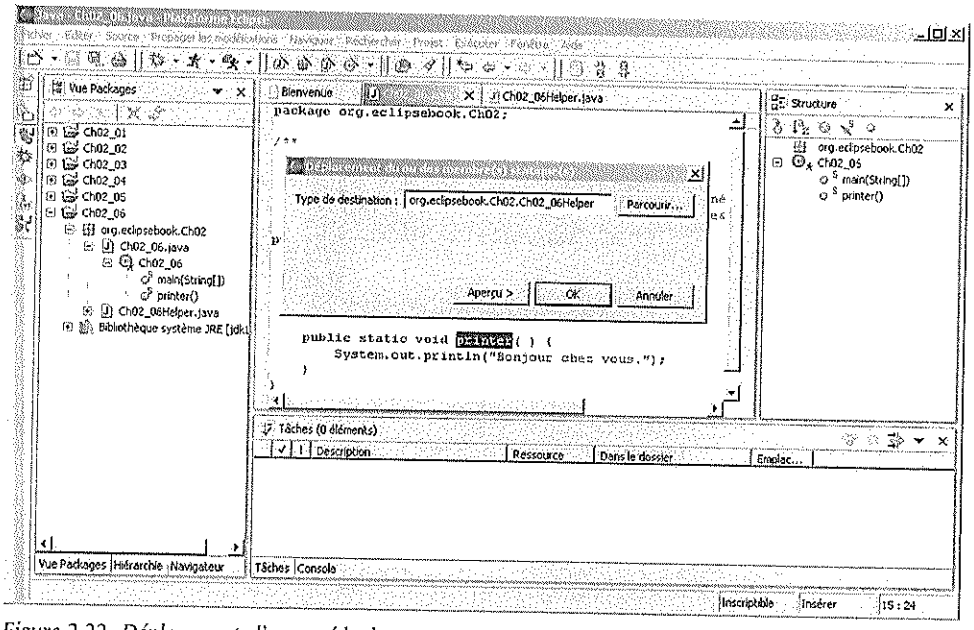


Figure 2-22. Déplacement d'une méthode vers une autre classe

```
        }

}
```

Extraction d'interfaces

On peut aussi extraire des interfaces au moyen de la propagation de modifications. À titre d'exemple, on va créer une interface pour la classe `Ch02_06Helper` (qui possède la méthode `printer`). On commence par convertir la méthode `printer` en méthode standard en enlevant le modificateur `static` dans le prototype de celle-ci. Ensuite, dans l'éditeur, avec un clic droit sur le nom de la classe `Ch02_06Helper` on sélectionne Propager les modifications → Extraire une interface. Cela ouvre la boîte de dialogue Extraction d'interface présentée dans la figure figure 2-23. Il faut sélectionner la méthode `printer` pour l'ajouter à la nouvelle interface, dont on donne alors le nom – `Ch02_06HelperInterface` – et cliquer sur le bouton OK.

Un nouveau fichier, *Ch02_06HelperInterface.java*, est alors créé, dans lequel est déclarée l'interface :

```
package org.eclipsebook.Ch02;

/**
 * @author Steven Holzner
 *
 * Pour changer le modèle de ce commentaire de type généré, allez à :
 * Fenêtre>Préférences>Java>Génération de code>Code et commentaires
 */
```

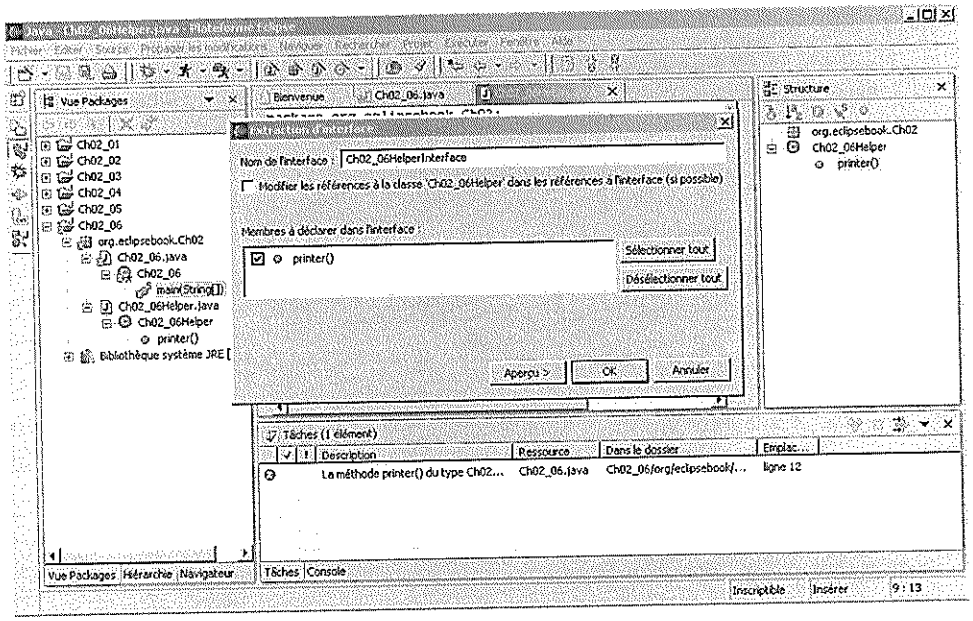


Figure 2-23. Extraction d'une interface

```
public interface Ch02_06HelperInterface {
        public abstract void printer();
}
```

Le code de la classe originelle est alors modifié, dans le sens où celle-ci déclare implémenter la nouvelle interface `Ch02_06HelperInterface`:

```
package org.eclipsebook.Ch02;

/**
 * @author Steven Holzner
 *
 * Pour changer le modèle de ce commentaire de type généré, allez à :
 * Fenêtre>Préférences>Java>Génération de code>Code et commentaires
 */
public class Ch02_06Helper implements Ch02_06HelperInterface {

        public void printer() {
                System.out.println("Bonjour chez vous.");
        }
}
```

En marge du renommage et de l'extraction d'interfaces, la propagation des modifications permet d'effectuer d'autres opérations, telles que la conversion de classes anonymes et de classes imbriquées, le changement de la signature d'une méthode et la conversion d'une variable locale en donnée membre d'une classe, etc. Le lecteur explorera avec profit les options du menu Propager les modifications.

Fonctionnalités complémentaires

Il est bon de connaître certaines fonctionnalités complémentaires. Par exemple, on sélectionne un lexème dans l'éditeur JDT, puis on clique sur le bouton droit de la souris et enfin on sélectionne l'option Ouvrir une déclaration, la déclaration de cette variable ou de cette méthode va apparaître. Cela permet de retrouver très facilement la façon dont celles-ci ont été créées. Diverses autres fonctionnalités complémentaires intéressantes sont présentées ci-dessous.

Visualisation d'une hiérarchie de type

Le menu contextuel de l'éditeur JDT propose une autre option, intitulée Ouvrir la hiérarchie des types ; lorsque celle-ci est cliquée par le programmeur, après qu'il ait sélectionné un objet, il obtient la hiérarchie du type de cet objet, dans la vue Hiérarchie de la perspective Java, comme on peut le voir dans partie gauche de la figure 2-24.

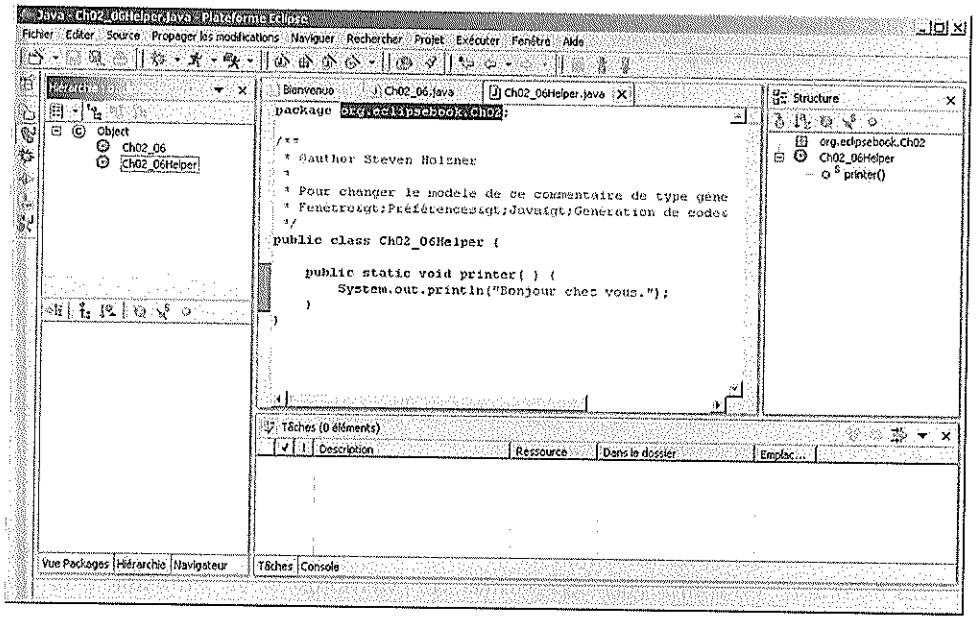


Figure 2-24. La vue Hiérarchie

Cette vue joue le rôle d'explorateur d'objets. Elle permet de visualiser la hiérarchie Java complète d'un type et d'accéder à la définition d'un membre quelconque de cette hiérarchie en double-cliquant sur celui-ci. Cette fonctionnalité est très utile lorsque le programmeur veut examiner tous les membres d'un objet, de la classe `System.out` par exemple – il lui suffit de sélectionner celle-ci dans son code et de cliquer sur l'option de menu Naviguer → Ouvrir un type de la hiérarchie.

La vue Hiérarchie est sensiblement identique à la vue Structure, visible dans la partie droite de la figure 2-24. Cependant, la vue Structure est conçue pour montrer une hiérarchie automatique du code développé plutôt qu'une hiérarchie de type spécifi-

quement demandée par le programmeur. La vue Structure est réactualisée automatiquement pour montrer la hiérarchie du type sur lequel travaille le programmeur à l'instant considéré, dans l'éditeur JDT.

Exploration de code Java

Il existe même une perspective entièrement dédiée à une exploration de projets orientée Java : la perspective Navigation Java. Pour ouvrir cette perspective, il suffit de sélectionner Fenêtre → Ouvrir la perspective → Navigation Java, ce qui donne le résultat montré dans la figure 2-25.

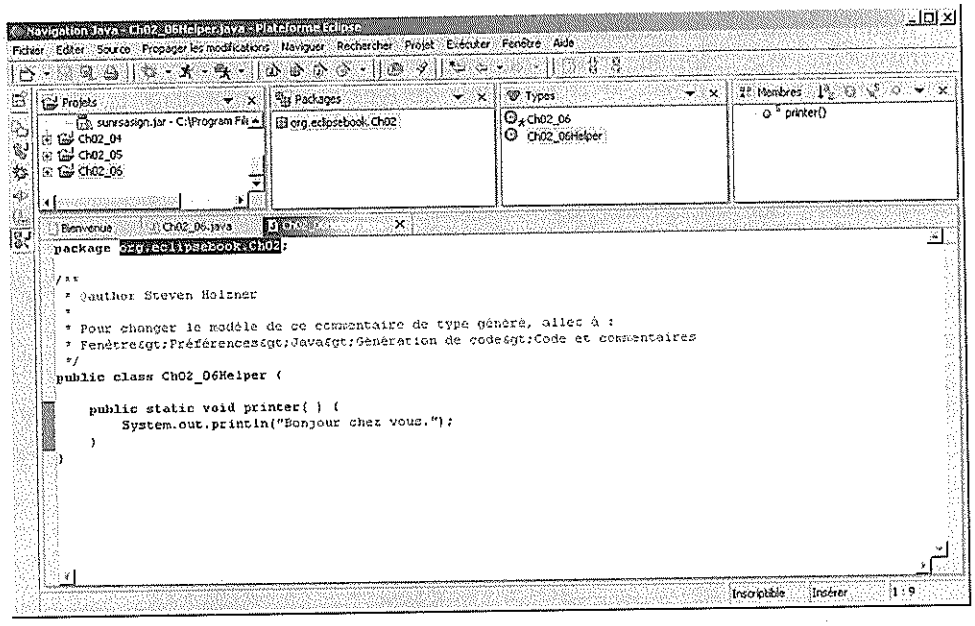


Figure 2-25. La perspective Navigation Java

Cette perspective présente l'information visualisée dans la perspective Java, mais ceci d'une nouvelle façon : les informations relatives aux types et aux membres sont affichées dans deux nouvelles vues séparées (la vue Membres est ici semblable à la vue Structure de la perspective standard Java). Comme on pouvait s'y attendre, les vues de cette perspectives sont coordonnées entre elles – sélectionner un élément dans la vue Projets fait apparaître ses paquetages dans la vue du même nom, par exemple. Sélectionner un élément dans la vue Membres fait apparaître celui-ci dans l'éditeur, etc.

Recherche de code

Eclipse propose aussi des outils très puissants de recherche dans du code, accessibles dans le menu Rechercher. En réalité, les principales options du menu Rechercher, Rechercher → Rechercher, Rechercher → Fichier, Rechercher → Aide, et Rechercher → Java, ouvrent toutes la même boîte de dialogue Recherche, bien que chaque option corresponde à un onglet différent de celle-ci. La recherche est particulièrement effi-

cace sous Eclipse – par exemple, la recherche peut se faire dans plusieurs fichiers et s'appuyer sur des jokers.

L'onglet Recherche d'un fichier de la boîte de dialogue Recherche permet d'effectuer des recherches dans plusieurs fichiers. Dans la figure 2-26, par exemple, on cherche le mot `main` dans tous les fichiers *.java* de l'espace de travail. Le champ d'action de cette recherche est défini au moyen des boutons radio du cadre Portée, le choix par défaut englobant tous les fichiers idoines de l'espace de travail (c'est-à-dire tous ceux des projets figurant dans la Vue Packages). Il est possible de restreindre la recherche à un ensemble de projets via un Jeu de documents (voir plus loin). Pour effectuer la recherche dans le seul projet courant ou dans un nombre restreint de projets, il suffit de sélectionner celui ou ceux de son choix dans la Vue Packages, d'ouvrir la boîte de dialogue Recherche, et de cliquer le bouton radio Ressources sélectionnées dans le cadre Portée.

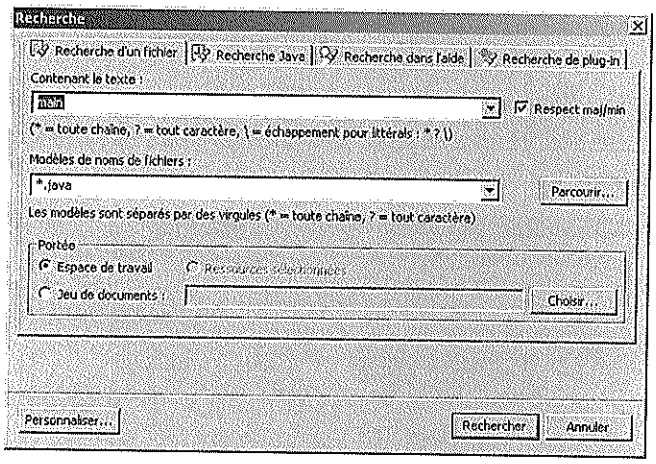


Figure 2-26. Recherche dans un ou pluseurs fichier(s)

Cliquer le bouton Rechercher déclenche la recherche du texte tapé, les résultats s'affichant dans la vue Recherche, qui apparaît au bas de la figure 2-27. En double-cliquant sur une des occurrences trouvées, le programmeur accède au code concerné qui apparaît dans l'éditeur JDT; on peut noter dans la figure les éléments en surbrillance et dans le code l'occurrence pointée par une flèche.

On peut aussi effectuer des recherches au moyen de l'onglet Java de la boîte de dialogue Recherche. Celui-ci permet de rechercher des éléments Java par type précis – types, méthodes, paquetages, constructeurs et membres –, comme on peut le voir dans la figure 2-28. On peut restreindre la recherche à des déclarations ou à des références seulement.

La possibilité d'effectuer des recherches dans des fichiers et dans des classes est l'un des gros avantages d'un EDI – ceux qui ont l'habitude d'utiliser un éditeur de texte et javac pour développer du code Java verront leur contexte de travail transcendé par Eclipse et ses outils de gestion de projet.

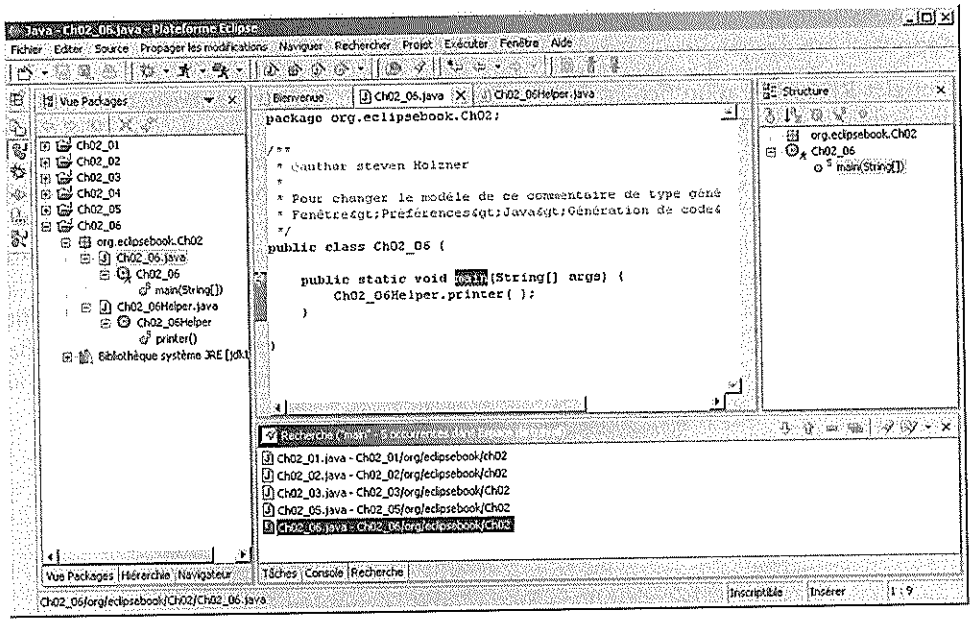


Figure 2-27. Visualisation des résultats d'une recherche

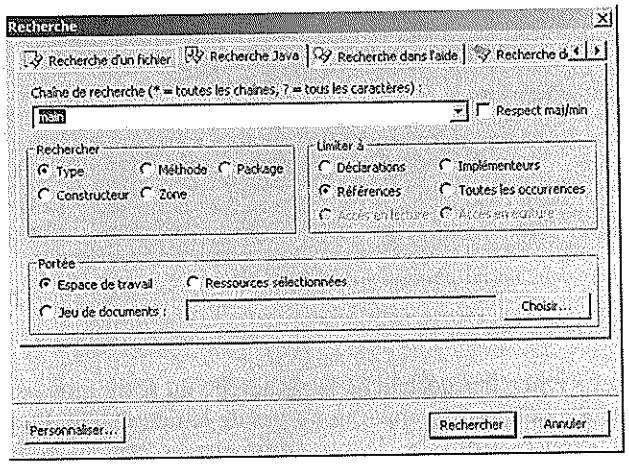


Figure 2-28. Recherche d'éléments Java

Personnalisation de l'environnement de développement

Le dernier thème abordé dans ce chapitre concerne la personnalisation de l'environnement de développement. Il est très facile de personnaliser Eclipse, en particulier pour les adaptations les plus simples – le programmeur peut déplacer n'importe quelle vue, n'importe quel éditeur ou n'importe quelle barre d'outil, de façon très naturelle au moyen de la souris.

Pour ceux qui n'aiment pas travailler dans un environnement dans lequel les différents composants d'interface peuvent bouger suite à un mouvement de souris malheureux, Eclipse propose l'option de menu Fenêtre → Verrouiller les barres d'outils pour en verrouiller les positions. De même, si les composants d'interface d'une perspective sont déplacés par erreur, l'option Fenêtre → Réinitialiser la perspective permet de les remettre en place.

On peut aussi personnaliser la façon dont Eclipse va générer du code. Par exemple, le style de génération de code par défaut est tel que les accolades ne sont pas placées sur une ligne dédiée :

```
public void printer() {
        System.out.println("Bonjour chez vous.");
}
```

Un programmeur ayant l'habitude de taper les accolades sur ses propres lignes souhaitera plutôt obtenir ceci :

```
public void printer()
{
        System.out.println("Bonjour chez vous.");
}
```

Cette préférence peut être imposée à Eclipse au moyen de l'option Fenêtre → Préférences, qui ouvre la boîte de dialogue Préférences de la figure 2-29. En choisissant l'option Java → Module de formatage de code, le programmeur peut spécifier des options de génération de code. Ici, il faut cliquer « Insertion d'une nouvelle ligne avant une accolade ouvrante » (voir figure) pour que les accolades soient placées dans des lignes dédiées : le code ci-dessous sera alors automatiquement adapté à la demande.

Voici une autre façon de personnaliser la génération de code par Eclipse. Lorsqu'un fichier est créé, le commentaire ci-dessous est inséré automatiquement :

```
/*
 * Créé le 7 juin 04
 *
 * Pour changer le modèle de ce commentaire de type généré, allez à :
 * Fenêtre>Préférences>Java>Génération de code>Code et commentaires
 */
```

Comme l'indique le texte du commentaire ci-dessus, il est possible de modifier celui-ci en utilisant l'option de menu Fenêtre → Préférences et de l'option Java → Génération de code → Code et commentaires (le > que l'on peut observer dans le code est la version HTML de >, qui est utilisée pour que la documentation Javadoc soit ouverte dans un navigateur ; le commentaire concerné est « Pour modifier le modèle de fichier généré, utiliser Fenêtre → Préférences → Java → Génération de code → Code et commentaires »). En ouvrant la boîte de dialogue Préférences et en sélectionnant l'option Java → Génération de code → Code et commentaires → Code → Nouveaux fichiers Java (voir figure 2-30), on peut éditer le modèle utilisé par Eclipse pour générer ce commentaire.

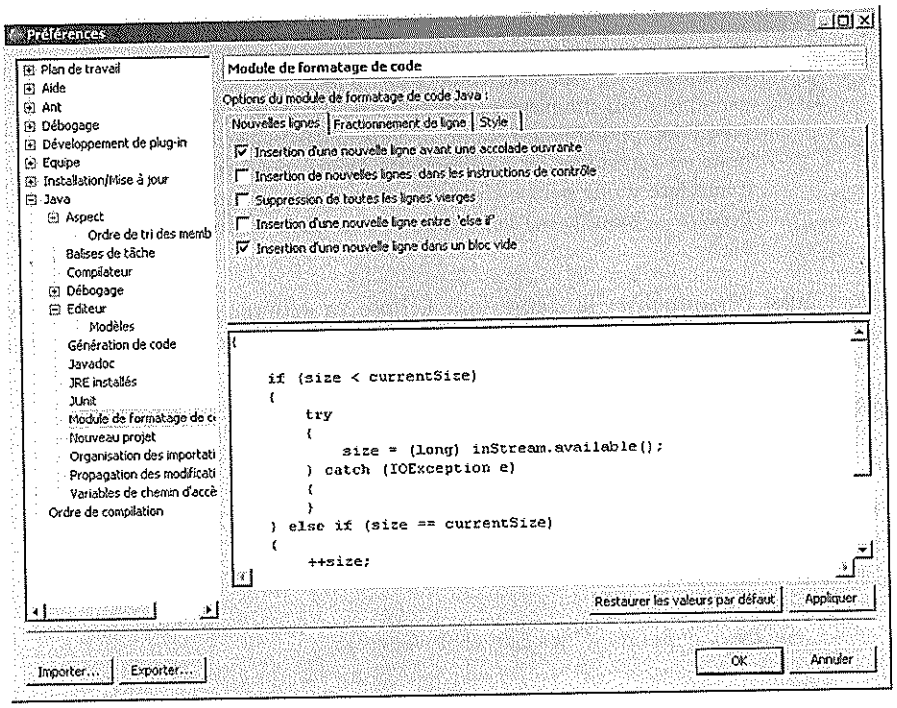


Figure 2-29. Personnalisation de la génération de code

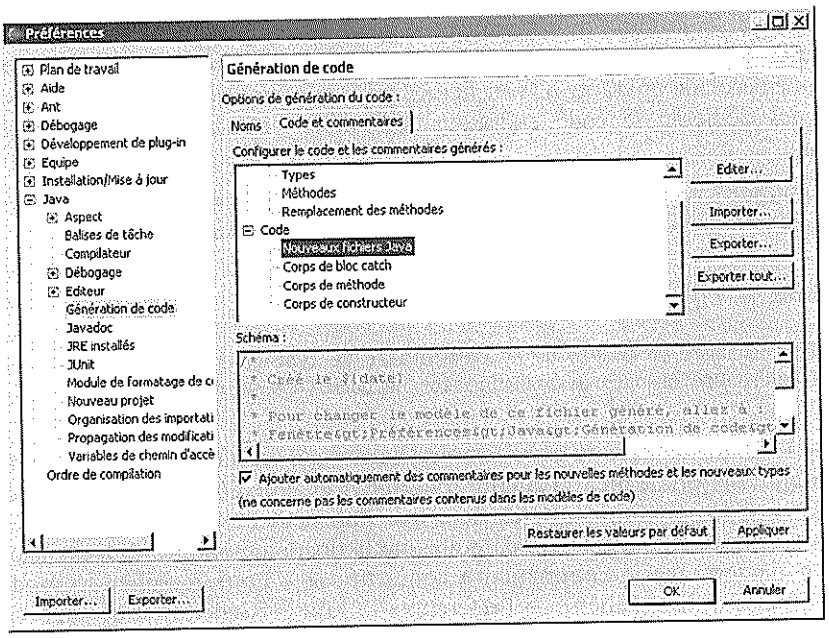


Figure 2-30. Configuration des modèles de commentaire

Il est même possible de créer de nouvelles options de l'assistant de code. Imaginons par exemple que l'on veuille créer un raccourci pour le code nécessaire à l'affichage de la date courante. Pour ce faire, il faut utiliser l'option Fenêtre → Préférences, et sélectionner Java → Editeur → Modèles. On crée alors un raccourci nommé ptd (voir figure 2-31). Dans cet exemple, le modèle `System.out.println("${date}");` sera remplacé par le code nécessaire à l'affichage de la date. Il est possible d'utiliser des valeurs autres que `${date}`, par exemple `${cursor}`, pour indiquer où l'éditeur doit placer le curseur une fois l'insertion effectuée (lorsqu'on tape `${`, l'assistant de code affiche toutes les valeurs utilisables dans les expressions de l'assistant de code).

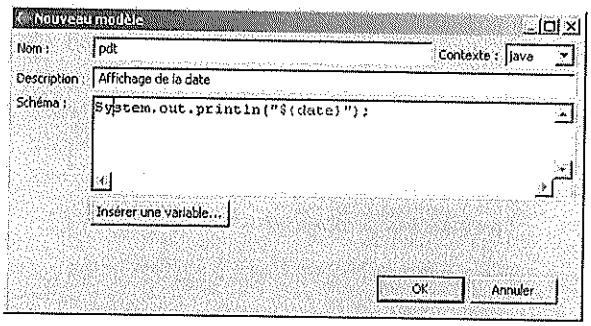


Figure 2-31. Création d'un raccourci pour l'assistant de code

Lorsqu'on tapera `ptd` dans le code, puis Ctrl+Espace, l'assistant de code insèrera le code associé (voir figure 2-32). En fait, si l'on tape `p` puis Ctrl+Espace, l'assistant de code va lister toute les options commençant par le caractère `p` – en particulier, `ptd`.

Une autre façon de personnaliser l'environnement de développement est de créer des *jeux de documents*. Un jeu de documents permet de restreindre les éléments apparaissant dans la Vue Packages. Par exemple, pour créer un jeu de documents ne comportant que les projets `Ch02_01` et `Ch02_02`, on ouvre le menu déroulant de la Vue Packages (apparaissant sous la forme d'un triangle renversé au sommet de cette vue) et on sélectionne l'option Sélectionner un jeu de documents, qui ouvre la boîte de dialogue Sélection d'un jeu de documents. Pour créer un nouveau jeu de documents, on clique sur le bouton Nouveau et on sélectionne le type Java, dans la boîte de dialogue Nouveau jeu de documents. On clique sur Suivant : comme on peut le voir dans la figure 2-33 on sélectionne les documents avec une coche. Dans le cas présent, le jeu de documents est nommé `1And2Only` et on sélectionne les projets `Ch02_01` et `Ch02_02`.

En cliquant Fin, on crée ce nouveau jeu de documents et on ouvre la boîte de dialogue Sélection d'un jeu de documents ; avec la sélection `1And2Only` et le clic sur OK, seuls les deux projets `Ch02_01` et `Ch02_02` apparaissent alors dans la Vue Packages (voir figure 2-34).

Pour faire réapparaître tous les projets dans la Vue Packages, il suffit de cliquer l'option Désélectionner le jeu de documents du menu déroulant de la Vue Packages.

En réalité, il est possible de personnaliser l'ensemble des perspectives. Pour cela, il suffit de sélectionner l'option de menu Fenêtre → Personnaliser la perspective, qui ouvre la boîte de dialogue Personnaliser la perspective (voir la figure 2-35). Dans cette

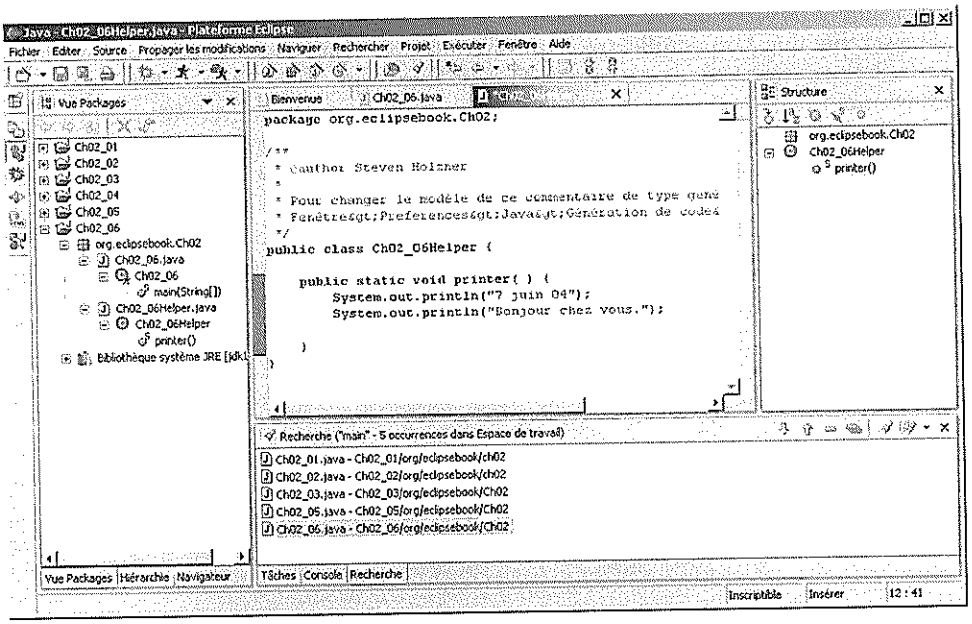


Figure 2-32. Personnalisation de l'assistant de code

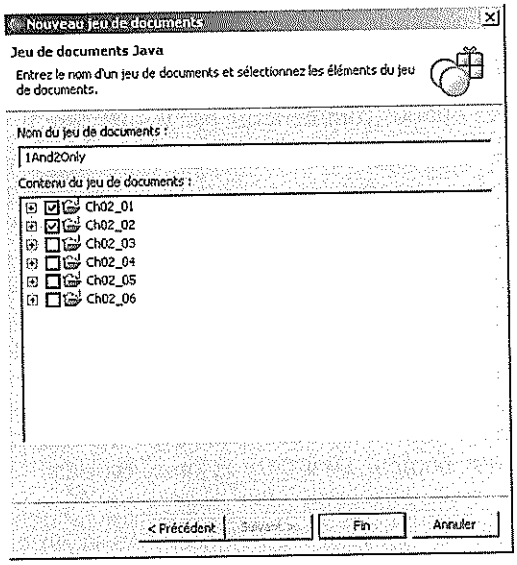


Figure 2-33. Création d'un jeu de documents

boîte, il est possible de personnaliser les options de menu disponibles pour la perspective courante ; par exemple, il est possible de spécifier vers quelles vues le programmeur peut basculer au moyen de l'option de menu Fenêtre → Afficher la vue, comme on peut le voir dans la figure.

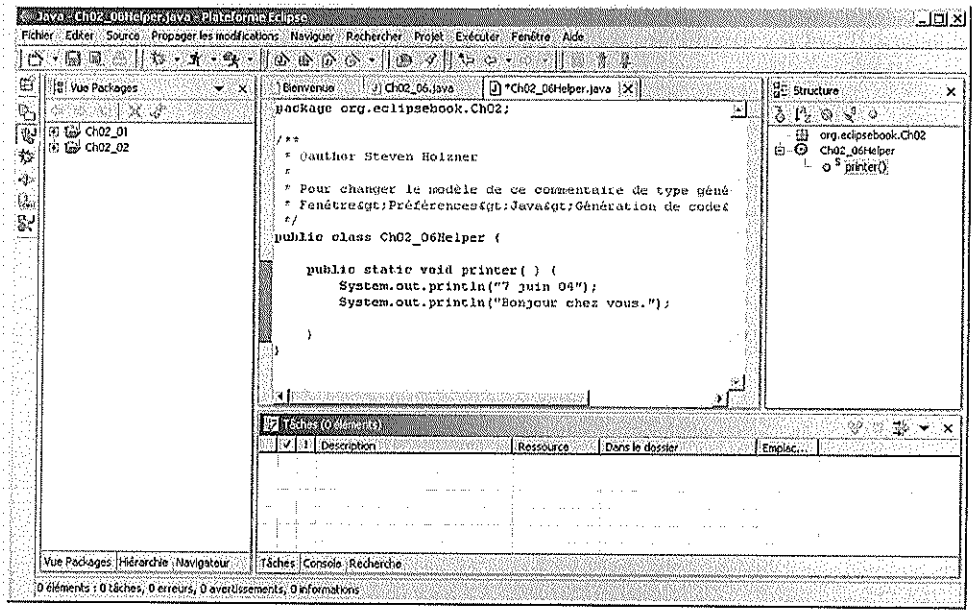


Figure 2-34. Utilisation d'un jeu de documents

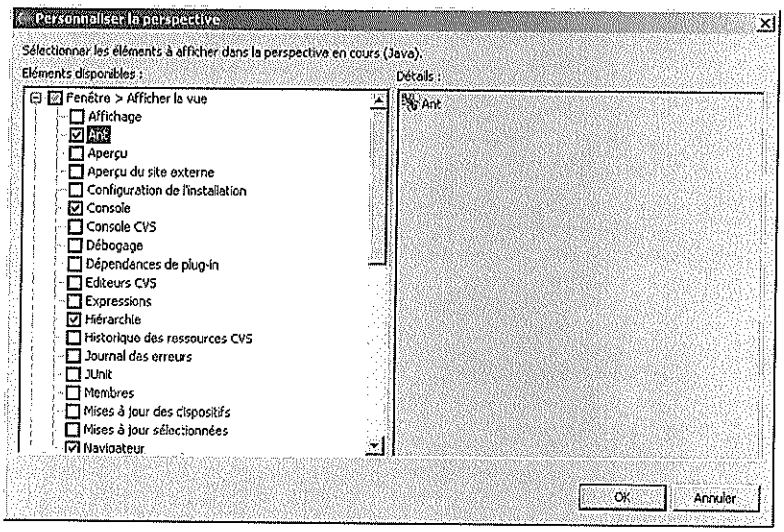


Figure 2-35. Personnalisation d'une perspective

Une fois les préférences du programmeur spécifiées, celui-ci peut les exporter afin de les mettre à la disposition d'autres programmeurs. Il suffit pour cela d'utiliser les boutons Importer et Exporter de la boîte de dialogue Fenêtre → Préférences.

Nous avons terminé la présentation des principales possibilités offertes par l'éditeur d'Eclipse. Dans le chapitre suivant seront présentés les outils de test et de deboguage du code Java.

CHAPITRE 3

Test de bon fonctionnement d'une application et débogage

La pratique des tests de bon fonctionnement et le débogage constituent deux facettes essentielles du développement d'applications Java. Eclipse fournit de nombreuses possibilités de test et de débogage, notamment dans le cadre de JUnit.

Test de bon fonctionnement d'applications Java : l'outil JUnit

JUnit est un open source destiné à tester des applications développées sous Eclipse. Il est possible de créer des classes basées sur l'outil JUnit et des classes normales dans le même projet : le code JUnit servira à tester les autres classes du projet. JUnit permet ainsi de construire un ensemble de jeux de tests pour tout programmeur travaillant sur une application : après chaque modification de code, quelques clics suffiront pour vérifier si l'application passe avec succès les jeux de tests créés au moyen de JUnit.

JUnit est conçu pour tester du code au moyen d'assertions permettant de tester des situations d'exécution variées :

`assertEquals(a, b)`
: Cette fonction teste si a est égal à b (a et b sont ou bien des valeurs primitives ou bien des objets possédant une méthode equals permettant d'effectuer des comparaisons)

`assertFalse(a)`
: Cette fonction teste si a est faux, où a est une valeur booléenne

`assertNotNull(a)`
: Cette fonction teste si a est différent de null, où a est ou bien un objet ou bien null

`assertNotSame(a, b)`
: Cette fonction teste si a et b ne se réfèrent pas au même objet

`assertNull(a)`
: Cette fonction teste si a est null, où a est ou bien un objet ou bien null

`assertSame(a, b)`
Cette fonction teste si a et b se réfèrent au même objet

`assertTrue(a)`
Cette fonction teste si a est vrai, où a est une valeur booléenne

Des séquences de tests JUnit sont ainsi construites au moyen des méthodes ci-dessus. Lorsqu'une application JUnit s'exécute, elle ouvre sa propre vue, qui donne au programmeur des informations en temps réel sur les tests qui ont été passés avec succès et sur ceux qui ont débouché sur un échec.

Création d'une application de test

On va illustrer le fonctionnement de JUnit sur la base de l'exemple `Ch03_01`. Cette application a pour fonction de remplir un tableau de nombres entiers et de permettre l'accès à ces valeurs au moyen des méthodes `get` et `set`.

En particulier, trois méthodes vont être construites : les valeurs retournées par celles-ci seront testées par JUnit :

`allocate`
Cette fonction alloue un tableau d'entiers et initialise ses éléments ; elle retourne une référence sur le tableau qui vient d'être alloué.

`set`
Cette fonction affecte une valeur à un des éléments du tableau ; la fonction retourne true si l'opération a réussi, false sinon.

`get`
Cette fonction lit une valeur dans le tableau d'entiers ; la fonction retourne cette valeur.

L'exemple 3-1 présente une première version du code.

Exemple 3-1. L'exemple Ch03_01.java

```
package org.eclipsebook.ch03;

public class Ch03_01 {

        private int[] array;

        public int[] allocate() {
                array[0] = 0;
                array[1] = 1;
                array[2] = 2;
                return array;
        }

        public int get(int index) {
                return array[index];
        }

        public boolean set(int index, int value) {
```

Exemple 3-1. L'exemple Ch03_01.java (suite)

```
                if (index < array.length && index >= 0) {
                        array[index] = value;
                        return true;
                } else {
                        return false;
                }
        }
}
```

La méthode allocate alloue un tableau d'entiers, initialise ses éléments et retourne une référence sur ce tableau :

```
private int[] array;

public int[] allocate() {
        array[0] = 0;
        array[1] = 1;
        array[2] = 2;
        return array;
}
```

La méthode get lit une valeur entière dans l'élément d'indice index du tableau et retourne celle-ci :

```
public int get(int index) {
        return array[index];
}
```

La méthode set affecte la valeur value à l'élément d'indice index du tableau et retourne true si l'affectation a réussi, false sinon :

```
public boolean set(int index, int value) {
        if (index < array.length && index >= 0) {
                array[index] = value;
                return true;
        } else {
                return false;
        }
}
```

L'étape suivante consiste à tester toutes ces méthodes – allocate, set et get – au moyen de JUnit.

Installation de JUnit

Pour travailler avec JUnit (voir *http://www.junit.org* pour plus de détails), il faut ajouter *junit.jar* au classpath du projet. Dans le cas présent, on va créer une nouvelle variable de classpath, JUNIT. Pour cela, il faut sélectionner l'option de menu Fenêtre → Préférences, qui ouvre la boîte de dialogue Préférences, puis cliquer l'option Java et sélectionner la sous-option Variables de chemin d'accès aux classes. Il faut ensuite cliquer le bouton Nouveau pour ouvrir la boîte de dialogue Nouvelle entrée de variable ; dans celle-ci, il faut taper le nom de la nouvelle variable, JUNIT, ainsi que le fichier correspondant, *junit.jar*, en indiquant son chemin d'accès complet. Pour la version 2.1.1 d'Eclipse, le chemin d'accès complet à ce fichier est *eclipse/plugins/org.junit_3.8.1/junit.jar*. On clique enfin sur OK.

Comme c'est souvent le cas sous Eclipse, une même tâche peut être effectuée de différentes façons : en voici une autre permettant d'ajouter une variable de classpath. On clique le bouton droit de la souris sur le projet `Ch03_01` et on sélectionne l'option Propriétés pour ouvrir la boîte de dialogue du même nom concernant `Ch03_01`. On sélectionne l'option Chemin de compilation Java et l'onglet Bibliothèques puis on clique sur le bouton Ajouter une variable pour ouvrir la boîte de dialogue Nouvelle entrée du chemin de variables. On clique ensuite sur le bouton Editer pour ouvrir la boîte de dialogue Préférences, puis sur le bouton Nouveau pour ouvrir la boîte de dialogue Nouvelle entrée de variable. On tape le nom de la nouvelle variable, `JUNIT`, puis le chemin d'accès complet à *junit.jar*. On clique enfin sur OK.

Pour le débogage, il peut s'avérer utile d'intégrer le code source du paquetage JUnit (cette opération est optionnelle ; on n'utilisera pas le code source JUnit dans les exemples qui suivent). On trouvera les fichiers JAR contenant le code source de nombreux paquetages dans le répertoire *eclipse/plugins/org.eclipse.jdt.source_2.1.1/src*. Pour créer une nouvelle variable associée au source JUnit, il suffit de reprendre la démarche décrite plus haut pour la création de `JUNIT`, mais en créant ici `JUNIT_SRC` (on ajoute en général le suffixe `_SRC` pour les variables de code source), et en l'associant au fichier *eclipse/plugins/org.eclipse.jdt.source_2.1.1/src/org.junit_3.8.1/junitsrc.zip*. Après avoir créé cette variable, il faut cliquer le bouton droit de la souris sur le projet `Ch03_01` et sélectionner l'option Propriétés : la boîte de dialogue Propriétés de `Ch03_01` est ouverte. On clique ensuite de l'option Chemin de compilation Java et l'onglet Bibliothèques ; on clique enfin le nœud `JUNIT`, comme on peut le voir dans la figure 3-1.

Lorsqu'on ouvre ainsi un nœud sur fichier JAR, on peut spécifier la localisation du code source et du document Javadoc associés. Pour lier la variable `JUNIT_SRC` au code source, on sélectionne l'option Connexion de la source et l'on clique sur Editer, ce qui ouvre la boîte de dialogue Configuration de la connexion de la source. On clique sur le bouton Variable, puis on double clique sur `JUNIT_SRC`, et enfin on presse le bouton OK pour fermer la boîte de dialogue. Le résultat de toutes ces manipulations est présenté dans la figure 3-2, qui montre que l'on a rendu le code source JUnit accessible à Eclipse. On clique finalement sur OK pour fermer la boîte de dialogue Propriétés du projet `Ch03_01`.

Tester une application avec JUnit

On peut commencer à tester l'application `Ch03_01`. Pour ce faire, on va créer une classe fille de la classe JUnit `TestCase` au moyen de l'assistant JUnit. Pour appeler cet assistant, il suffit de cliquer le bouton droit de la souris sur la classe `Ch03_01` dans la vue Packages et de sélectionner l'option Nouveau → Autre : la boîte de dialogue Nouveau de la figure figure 3-3 apparaît alors.

Arrivé à cette étape, on peut cliquer le nœud Java se trouvant dans la page de gauche et sélectionner l'entrée nommée JUnit. Dans la page de droite, on sélectionne l'entrée `Scénario de test` (associée à la classe `TestCase`) et l'on clique Suivant, ce qui provoque l'apparition de la page présentée dans la figure 3-4.

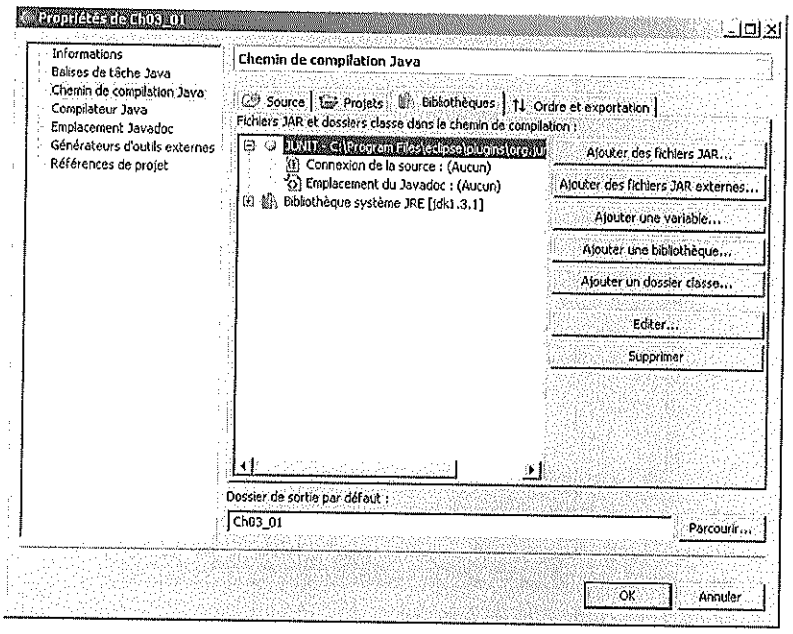


Figure 3-1. Accessibilité du code source

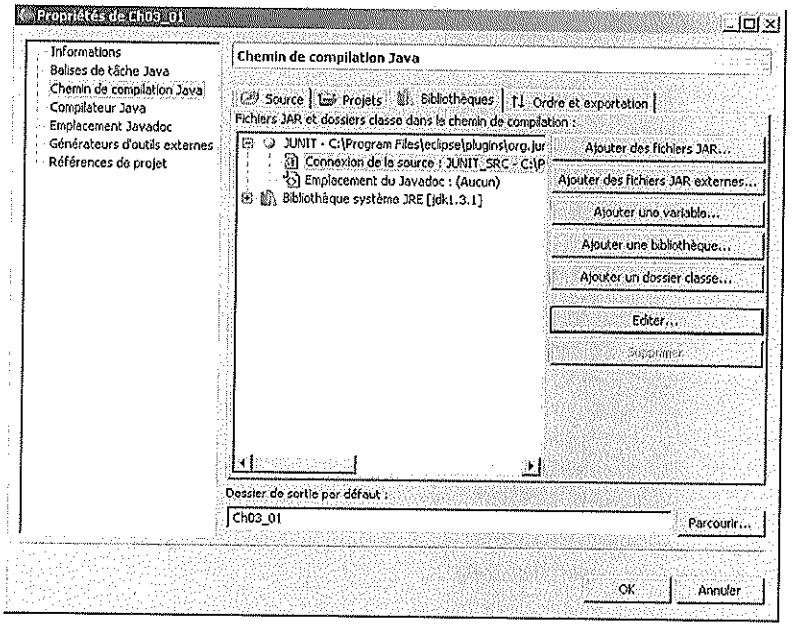


Figure 3-2. Utilisation de la variable JUNIT_SRC

C'est l'usage général de donner à une classe JUnit un nom qui est celui de la classe testée, complété par le mot `Test` : dans cette boîte de dialogue, on va donc taper `Ch03_01Test` dans la zone d'édition nommée Scénario de test. De plus, on va sélec-

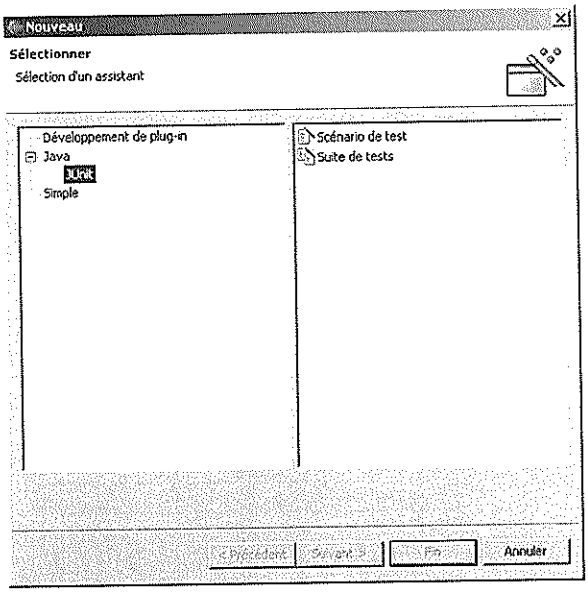


Figure 3-3. Création d'une classe fille de TestCase

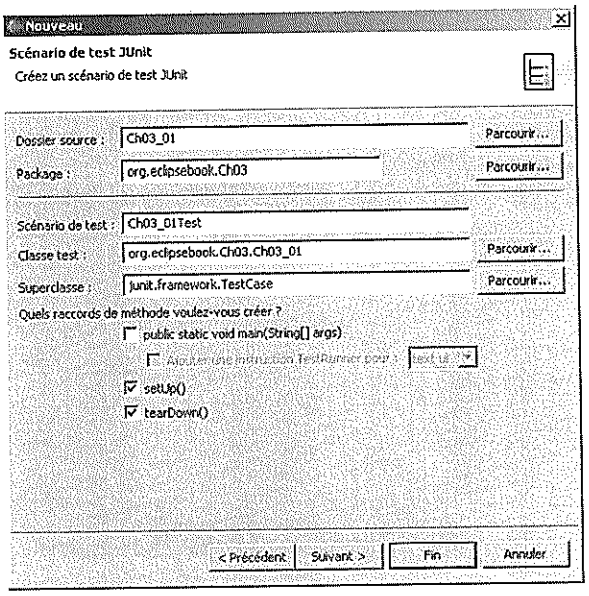


Figure 3-4. Création d'une application JUnit

tionner les méthodes `setUp` et `tearDown` (voir figure 3-4). Ces méthodes permettent au programmeur d'initialiser et de nettoyer les données et/ou les objets impliqués dans le scénario de test (en terminologie JUnit, on parle de *fixtures*). On clique alors sur Suivant pour ouvrir la page suivante, présentée dans la figure 3-5.

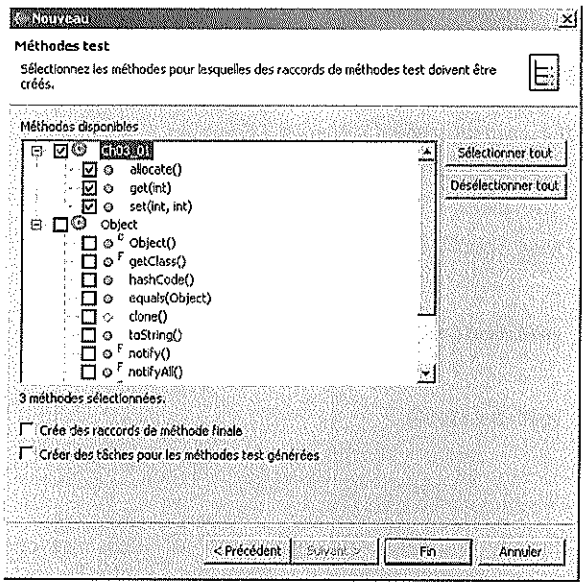


Figure 3-5. Sélection des méthodes à tester

On sélectionne dans cette page les méthodes à tester, ce qui permet à l'assistant JUnit de créer des stubs pour celles-ci. Dans l'exemple présent, on veut tester `allocate`, `set` et `get` : il faut donc les sélectionner, comme le montre la figure 3-5 et cliquer sur le bouton Fin pour créer la classe `Ch03_01Test` (voir exemple 3-2). Dans cette classe, on note la présence d'une méthode vide (stub) pour chacune des trois méthodes à tester (`allocate`, `set` et `get`) : `testAllocate`, `testSet` et `testGet`.

Exemple 3-2. L'exemple Ch03_01Test.java

```
import junit.framework.TestCase;
        .
        .
        .
public class Ch03_01Test extends TestCase {
        /**
         * Constructor for Ch03_01Test.
         * @param arg0
         */
        public Ch03_01Test(String arg0) {
                super(arg 0);
        }

        /*
         * @see TestCase#setUp()
         */
        protected void setUp() throws Exception {
                super.setUp();
        }
```

Exemple 3-2. L'exemple Ch03_01Test.java (suite)

```
        /*
         * @see TestCase#tearDown()
         */
        protected void tearDown() throws Exception {
                super.tearDown();
        }

        public void testAllocate() {
        }

        public void testGet() {
        }

        public void testSet() {
        }

}
```

L'étape suivante consiste à insérer du code dans ces stubs : ce code va appeler les méthodes `allocate`, `set`, et `get` de la classe `Ch03_01` de telle façon que des assertions JUnit soient appliquées sur les résultats obtenus. Une instance de la classe `Ch03_01` est nécessaire à l'appel de ces méthodes : on va la nommer `testObject`. Pour créer `testObject`, on utilise la méthode `setUp` du code JUnit. Cette méthode est appelée juste avant le lancement du test JUnit. Créons donc l'instance `testObject` de la classe à tester, `Ch03_01` :

```
Ch03_01 testObject;
    .
    .
    .
protected void setUp() throws Exception {
        super.setUp();
        testObject = new Ch03_01();
}
```

Nous allons pouvoir effectuer des tests sur cet objet. Par exemple, la méthode `allocate` est supposée créer un tableau d'entiers et retourner une référence sur ce tableau : on peut tester si cette référence est bien différente de `null` en utilisant `assertNotNull` dans `testAllocate` :

```
public void testAllocate() {
        assertNotNull(testObject.allocate());
}
```

La méthode `get` est supposée lire une valeur dans le tableau ; on peut effectuer des tests sur celle-ci grâce à la méthode `assertEquals`, en la comparant à une valeur particulière, dans le code de `testGet` :

```
public void testGet() {
        assertEquals(testObject.get(1), 1);
}
```

Enfin, la méthode set est supposée retourner true si son exécution s'est déroulée convenablement ; on peut la tester au moyen de assertTrue, de la façon suivante :

```
public void testSet() {
        assertTrue(testObject.set(2, 3));
}
```

Ayant tapé ce code, on peut sélectionner la classe Ch03_01Test dans la vue Packages et cliquer l'option de menu Exécuter en tant que → Test JUnit, ce qui ouvre la vue située à gauche de la figure 3-6.

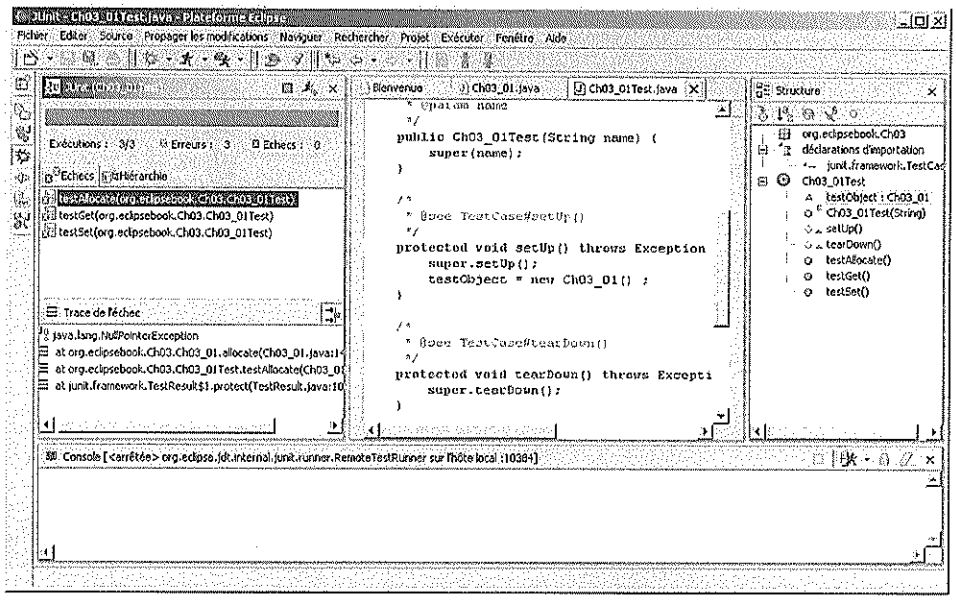


Figure 3-6. Exécution d'un test JUnit

Une barre rouge apparaît en haut et à gauche de cette vue (voir figure 3-6) ; celle-ci indique qu'il y a des problèmes. Comme on peut le voir en cliquant l'onglet Echecs de la vue JUnit, les trois tests sont marqués d'un X, qui signale un problème. Examinons le premier test, testAllocate, qui permet de dépister la valeur pathologique null comme valeur retournée par l'allocation de tableau :

```
public void testAllocate() {
        assertNotNull(testObject.allocate());
}
```

Un examen rapide de la méthode allocate révèle le problème – le code tente d'affecter des valeurs aux éléments du tableau sans avoir créé celui-ci au préalable. On résoud ce problème en ajoutant le code suivant dans la classe Ch03_01 :

```
private int[] array;
    .
    .
    .
public int[] allocate() {
```

```
        array = new int[3];
        array[0] = 0;
        array[1] = 1;
        array[2] = 2;
        return array;
    }
```

Exécutant à nouveau `Ch03_01Test`, on constate que seules les deux méthodes `testGet` et `testSet` échouent (voir figure 3-7).

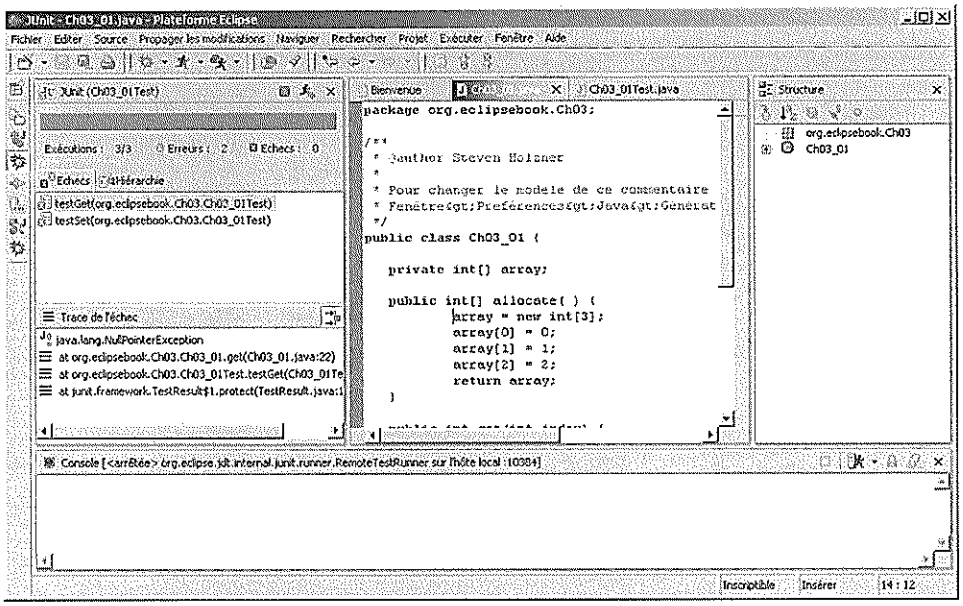


Figure 3-7. Deux tests échouent à présent

Que se passe-t-il avec `testGet` et `testSet` ? En fait, les deux tests JUnit sont effectués *indépendamment*. Cela signifie que bien que le premier test, `testAllocate`, appelle préalablement `allocate`, celle méthode n'est pas appelée par les deux autres, `testGet` et `testSet`. Pour initialiser le tableau pour les deux autres tests, il est nécessaire d'appeler explicitement `allocate` dans `testGet` et dans `testSet`. Voici le code effectuant ces appels :

```
public void testGet() {
        testObject.allocate();
        assertEquals(testObject.get(1), 1);
}

public void testSet() {
        testObject.allocate();
        assertTrue(testObject.set(2, 3));
}
```

Tous les tests sont alors passés avec succès par le code, comme on peut le voir dans la figure 3-8. La barre située au sommet de la vue JUnit est alors verte : Eclipse signifie ainsi au programmeur qu'il n'y a plus de problème.

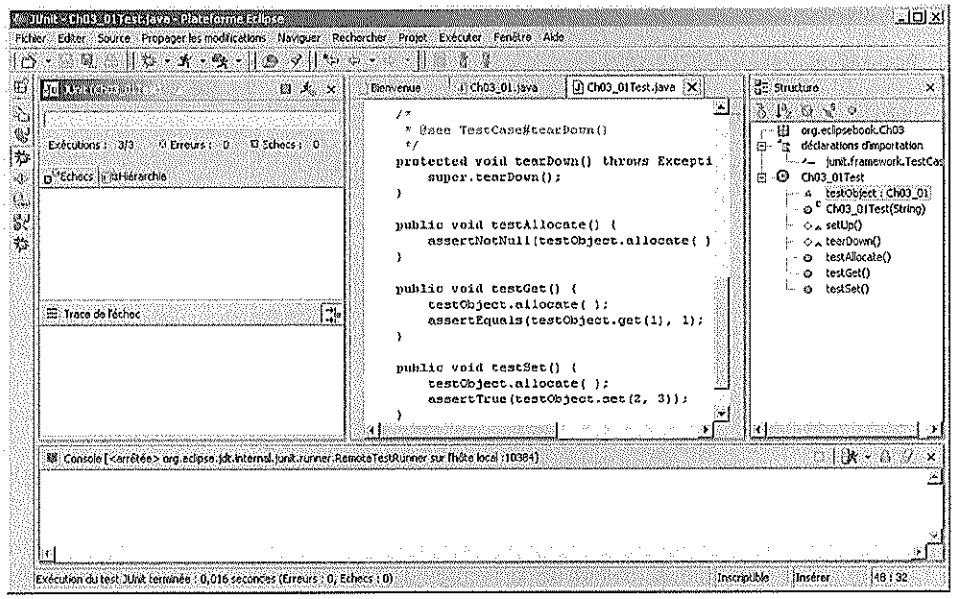


Figure 3-8. Les trois tests sont passés avec succès, à présent

JUnit fournit ainsi un moyen relativement simple pour créer des jeux de tests standard qui peuvent être conçus en quelques clics de souris. Une fois les tests créés, il suffit d'exécuter la classe JUnit ainsi créée. Cet outil est très utile car il permet à un ensemble de programmeurs d'appliquer les mêmes tests à un code qu'ils développent en commun.

Cependant, JUnit ne teste que la *conformité* à un jeu de tests donné – si un problème plus profond se pose, il faut alors *déboguer* le code.

Débogage

Les possibilités de débogage d'Eclipse sont très impressionnantes et sans commune mesure avec tous les autres EDI. Commençons par étudier un exemple comportant une erreur logique. On verra par la suite des fonctionnalités plus complexes, tels que la création de configurations de lancement du débogueur, la modification de code durant une session de débogage, l'arrêt de l'exécution d'un programme en boucle infinie, etc. La possibilité de déboguer interactivement du code est une exigence standard pour un EDI – ce qui constitue une formidable amélioration par rapport aux outils fournis par Java – mais Eclipse va bien au-delà de cette exigence.

Un programme bogué

Notre programme bogué figure dans l'exemple 3-3. Eclipse fonctionne particulièrement bien avec les *contextes d'exécution empilés*, qui contiennent l'ensemble des variables locales des appels successifs des méthodes. Pour illustrer les capacités du débogueur, on va mettre en place un exemple qui crée plusieurs contextes d'exécution empilés : le calcul d'une factorielle (par exemple, la factorielle de 6, notée 6!, est égale à 6 * 5 * 4 * 3 * 2 * 1 = 720), dont le résultat est obtenu par appel récursif.

Exemple 3-3. L'exemple Ch03_02.java

```
package org.eclipsebook.ch03;

public class Ch03_02 {

        public static void main(String[] args) {
                System.out.println(factorial(6));
        }

        public static int factorial(int value) {
                if(value == 0){
                        return value;
                }
                else {
                        return value * factorial(value - 1);
                }
        }
}
```

On calcule une factorielle de la manière suivante : `n! = n * (n - 1) * (n - 2) ... * 1`. Dans notre exemple, le calcul de la factorielle est effectué de façon récursive : la méthode `factorial` calcule `factorial(n)` en multipliant n par `factorial(n - 1)` ; cette fonction s'auto-appelle donc pour calculer la factorielle de (n-1). Pour déterminer cette dernière, la méthode `factorial` multiplie (n-1) par `factorial(n - 2)`, et ainsi de suite, jusqu'à ce que cette méthode arrive au calcul de la factorielle de 0, qui vaut 1. Voici le code effectuant le calcul de la factorielle :

```
public static int factorial(int value) {
        if(value == 0){
                return value;
        }
        else {
                return value * factorial(value - 1);
        }
}
```

Dans cet exemple, on tente de calculer la factorielle de 6 (notée 6!), qui vaut 720. Voici le code de la méthode `main` :

```
public static void main(String[] args) {
        System.out.println(factorial(6));
}
```

Malheureusement, la première exécution de cet exemple débouche sur le résultat 6! = 0, comme on peut le voir dans la vue Console de la figure 3-9.

Instauration d'un point d'arrêt

Aucune exception n'a été levée : il y a donc un problème de logique dans le code, qu'il convient de déboguer à présent. On va tout d'abord examiner le code en cours d'exécution, que l'on va suspendre à un endroit précis au moyen d'un point d'arrêt. Pour instaurer un tel point d'arrêt dans l'éditeur JDT, il suffit de double-cliquer la barre de marquage face à la ligne de code sur laquelle on veut le créer (on peut aussi sélectionner la ligne concernée et cliquer l'option de menu Exécuter → Ajouter/

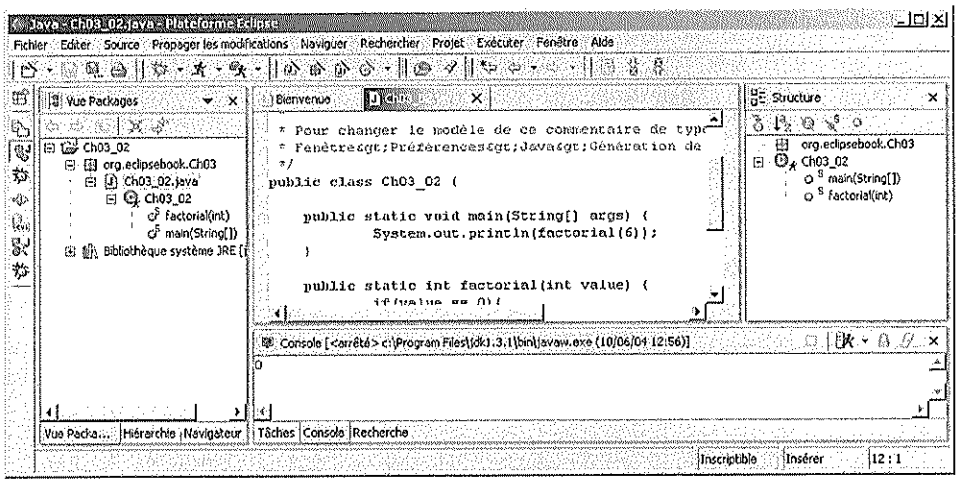


Figure 3-9. Première tentative d'exécution du programme bogué

supprimer un point d'arrêt). Pour enlever ce point d'arrêt, il suffit de double-cliquer sur celui-ci.

Dans le cas présent, on va créer un point d'arrêt sur la ligne `return value * factorial(value - 1)`, afin de comparer la valeur calculée après chaque appel récursif de la méthode `factorial`. Le nouveau point d'arrêt apparaît sous la forme d'un point bleu dans la barre de marquage (voir figure 3-10).

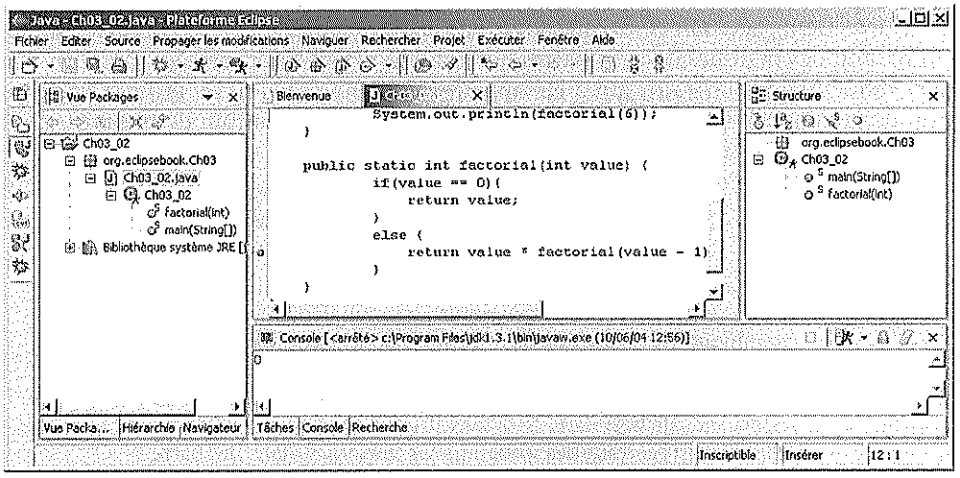


Figure 3-10. Instauration d'un point d'arrêt

Pour déboguer le code, on sélectionne l'option de menu Exécuter → Déboguer en tant que → Application Java, ce qui fait apparaître le perspective Débogage de la figure 3-11.

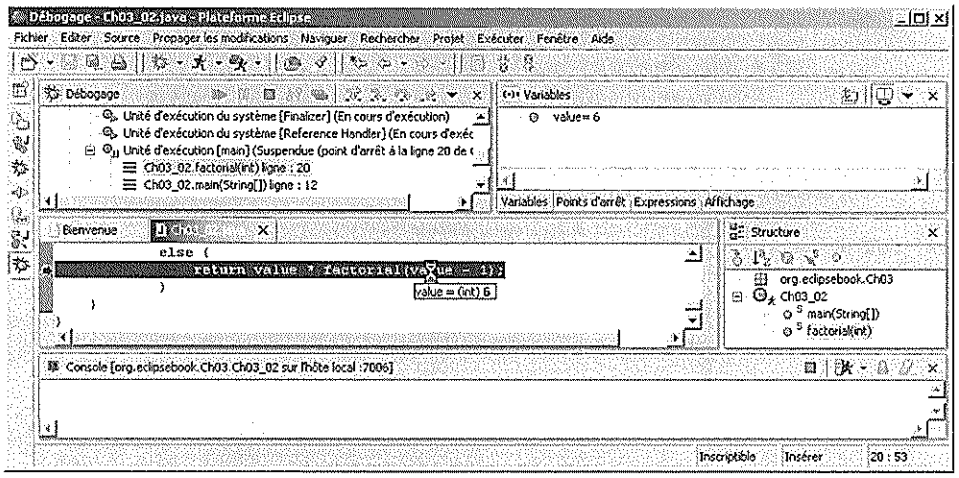
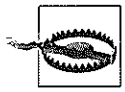


Figure 3-11. La perspective Débogage



Si l'on clique l'option de menu Exécuter → Exécuter, les points d'arrêt seront ignorés durant l'exécution.

Le code s'exécute jusqu'à ce que le point d'arrêt soit rencontré : l'exécution s'interrompt alors sur la ligne concernée, qui est marquée par une flèche dans l'éditeur du débogueur.

La perspective Débogage mérite d'être examinée en profondeur. On peut noter des options pour les programmes en cours de débogage dans la partie supérieure gauche de la vue Débogage. On remarquera en particulier la présence des contextes d'exécution empilés marqués de trois barres horizontales. Dans le cas présent, on se trouve dans la méthode `factorial`, qui a été appelée par la méthode `main`. Dans la barre de titre de la vue Débogage, on note la présence d'un certain nombre de boutons, dont les trois les plus à gauche sont, de la gauche vers la droite, Reprise (reprendre l'exécution du code à nouveau), Interrompt l'application (interrompre temporairement l'exécution du code) et Termine l'application (arrêt du débogage).

Les sessions successives de débogage sont ajoutées dans la vue Débogage, ce qui peut encombrer assez rapidement l'interface graphique (les trois dernières sessions sont accessibles, afin de permettre au programmeur d'étudier le comportement de son programme). Pour se débarrasser des précédentes sessions de débogage affichées dans cette vue, il suffit de cliquer sur le bouton Supprime tous les programmes terminés dans la barre d'outils de cette vue (le bouton contenant trois rectangles superposés).

À la droite de la vue Débogage se trouve un ensemble de vues empilées – Variables, Points d'arrêt, Expressions et Affichage. La vue Variables permet d'examiner le contenu des variables locales, qui peuvent de plus être éditées et modifiées en cours d'exécution (comme on le verra plus loin), ce qui permet de dépister interactivement

l'origine des problèmes d'exécution. Eclipse surveille aussi la modification des valeurs contenues dans les variables locales, qui changent alors de couleur à cet instant précis (elles apparaissent en rouge). La partie inférieur de la vue Variables est appelée *sous-fenêtre Détails* : elle permet une visualisation du contenu des variables selon un format plus détaillé

La vue Points d'arrêts permet de gérer les points d'arrêt présents dans le code; on y accède par un double-clic ou on sélectionne l'une des options du menu contextuel telle que Activer, Désactiver, Supprimer ou Tout supprimer.

La vue Expressions permet d'évaluer des expressions, comme on le verra dans quelques pages. Lorsqu'une expression est sélectionnée dans l'éditeur, il suffit de cliquer dessus avec le bouton droit de la souris et de sélectionner Inspecter – l'expression est alors évaluée dans la vue Expressions. De la même façon, sélectionner la vue Affichage dans le menu contextuel fait apparaître le résultat dans cette vue.

L'éditeur de la perspective Débogage est essentiellement le même que celui de la perspective Java ; il permet de visualiser le contenu d'une variable en plaçant simplement le curseur de la souris sur celle-ci. Dans la figure 3-11, par exemple, on examine la variable value, dont le contenu vaut 6, puisque la méthode `factorial` a été appelée avec cette valeur pour argument.

Dans la partie inférieure droite de l'interface graphique se trouve la vue Structure (voir aussi la figure 3-11), qui est la même que pour la perspective Java. Sous l'éditeur se trouve la vue Console, qui visualise les affichages exécutés par le programme, comme dans le cadre de la perspective Java.

Lorsqu'une une session de débogage est terminée, la perspective Débogage reste affichée. Le programmeur peut bien sûr basculer vers la perspective Java au moyen de l'option de menu Fenêtre → Ouvrir la perspective → Java, mais durant le cycle de débogage, il est plus facile de basculer d'une perspective à l'autre en cliquant sur les icônes raccourcis visibles à l'extrême gauche de la figure 3-11.

Exécution du code en pas à pas

La façon la plus simple d'exécuter un code temporairement interrompu est de procéder à l'exécution de celui-ci ligne après ligne. Eclipse offre à cet effet quatre possibilités, associées aux quatre boutons contenant des flèches, localisés dans la partie droite de la barre de titre de la vue Débogage (ces options sont aussi accessibles au moyen du menu Exécuter, lorsque l'exécution du code est temporairement suspendue) :

Avancer d'un pas avec filtres (Shift + F5)

Cette commande permet d'entrer dans le code d'une fonction appelée par le code, avec l'aide de filtres déterminés, sauf si le filtre de la méthode est inactif, auquel cas la fonction appelée est exécutée globalement (voir plus loin).

Avancer d'un pas avec entrée (F5)

Exécute l'instruction sélectionnée. S'il s'agit d'un appel de fonction, on entre dans le code de celle-ci, dont les lignes sont exécutées une par une.

Avancer d'un pas sans entrée (F6)

Cette commande exécute l'instruction sélectionnée. S'il s'agit d'un appel de fonction, celle-ci est exécutée globalement, sans que l'on entre dans le détail de celle-ci.

Exécuter jusqu'à l'instruction de retour (F7)

Cette commande provoque l'exécution de la fonction courante jusqu'à son terme, mais après le retour de la fonction, replace le programme en interruption temporaire (sauf si un point d'arrêt est rencontré avant le terme de la fonction).

Dans le présent exemple, l'exécution du programme est temporairement suspendue sur la ligne `return value * factorial(value - 1)`. En pressant le touche F5 (avancer d'un pas *avec entrée*), on va exécuter `factorial(value - 1)` en entrant dans la fonction (voir figure 3-12) ; la nouvelle valeur de la variable `value` est 5.

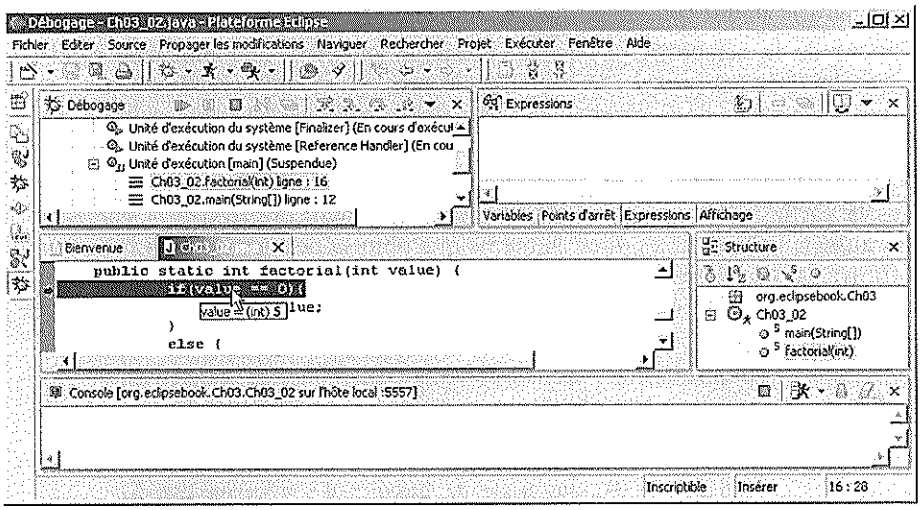


Figure 3-12. Exécution pas à pas

Jusqu'à présent, la méthode `factorial` s'exécute conformément à ce que l'on attend d'elle. Lors du premier appel, `value` valait 6 ; au deuxième appel de `factorial`, `value` vaut 5.

On peut utiliser aussi des *filtres de progression*, qui indiquent quelle portion de code le programmeur veut tester en pas à pas. Lorsque le programmeur utilise l'option Avancer d'un pas avec filtres, le code de la classe ou du paquetage faisant l'objet d'un filtrage n'est pas exécuté en mode pas à pas avec entrée, ce qui évite une exécution en mode pas à pas sur toute l'application et la restreint seulement à la section de code où elle est véritablement utile. Pour créer des filtres de progression, il faut cliquer l'option de menu Fenêtre → Préférences ; dans la boîte de dialogue Préférences ainsi ouverte, il suffit de sélectionner l'option Java → Débogage → Filtrage de progression (voir figure 3-13). Pour filtrer le code, on utilise les cases à cocher de la rubrique Filtres de progression définis ; on peut aussi créer de nouveaux filtres en cliquant sur le bouton Ajouter un filtre.

On pourrait exécuter l'ensemble du code pas à pas jusqu'à ce que l'on identifie le problème. On va plutôt utiliser le point d'arrêt déjà créé pour le trouver plus vite.

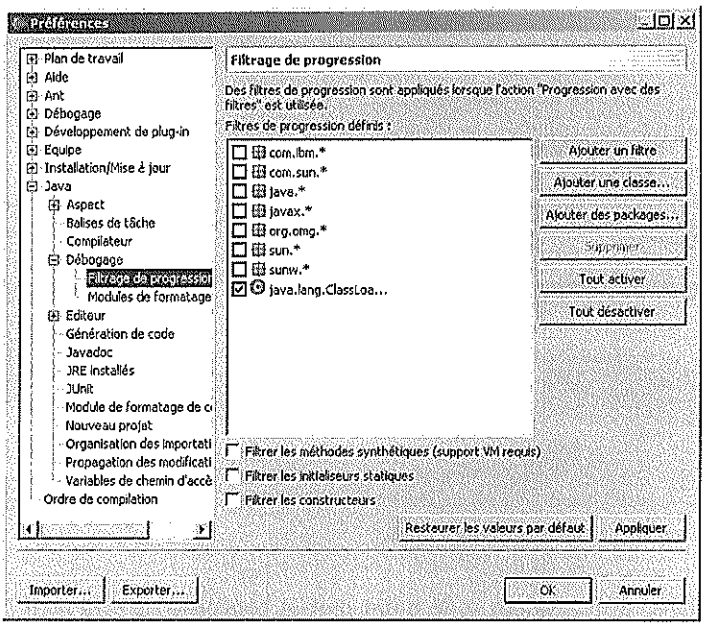


Figure 3-13. Création de filtres de progression

Reprise de l'exécution jusqu'au point d'arrêt suivant

On peut donc continuer à exécuter le code pas à pas, en entrant dans la fonction `factorial` à chaque itération, ce qui est un peu fastidieux. On va plutôt exécuter le code jusqu'au point d'arrêt suivant : pour ce faire, il suffit de cliquer le bouton Reprise, dans la vue Débogage (c'est le bouton le plus à gauche dans le groupe de boutons figurant dans la barre de titre de la vue Débogage).

Avant de faire cela, on peut demander à la perspective Débogage une visualisation permanente de la variable `value`, ce qui nous permettra d'inspecter sa valeur en continu. Pour cela, on sélectionne cette variable dans l'éditeur, on clique le bouton droit de la souris et on sélectionne l'option Inspecter. La variable `value` est alors affichée en permanence dans la vue Expressions.

On peut alors cliquer le bouton Reprise. L'exécution reprend jusqu'au point d'arrêt suivant, comme on peut le voir dans la figure 3-14. On est toujours dans le même appel de `factorial`, la variable `value` contenant toujours la valeur 5 (voir la vue Expressions).

Il suffit alors de cliquer le bouton Reprise pour reprendre l'exécution, itération après itération de `factorial`. Il existe cependant un moyen encore plus simple.

Création d'un compteur d'occurrences de points d'arrêt

Le calcul de la factorielle de 6 est censé déclencher six itérations de la méthode `factorial` : il faudrait donc cliquer six fois le bouton Reprise. On peut néanmoins configurer un point d'arrêt à l'aide d'un *compteur d'occurrences*. Lorsqu'un tel comp-

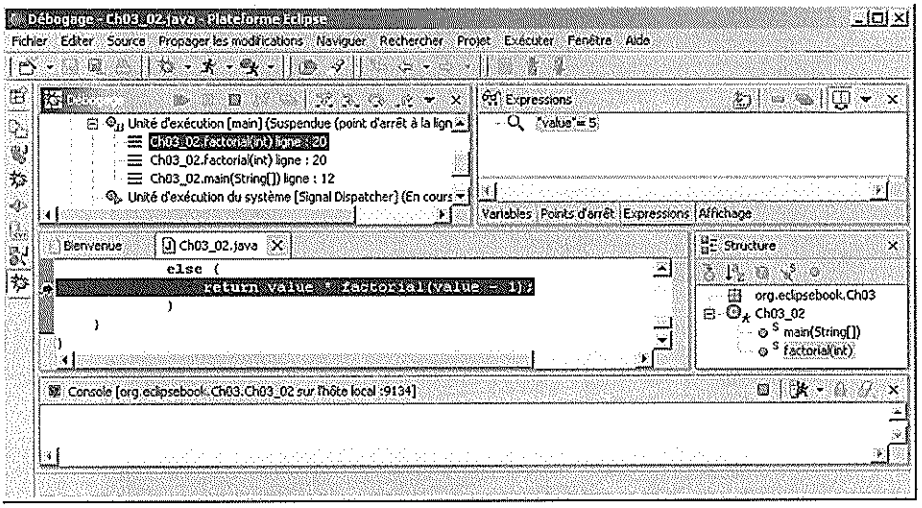


Figure 3-14. Exécution jusqu'à un point d'arrêt

teur est initialisé à une valeur *n*, le point d'arrêt concerné ne sera actif qu'au *n*-ème passage sur celui-ci.

Pour créer un compteur d'occurrences, il faut cliquer le bouton droit de la souris sur le point d'arrêt concerné dans la vue Points d'arrêt et sélectionner l'option Propriétés, ce qui ouvre la boîte de dialogue Propriétés de point d'arrêt de ligne Java, présentée dans la figure 3-15. Dans le cas présent, on coche la case à cocher Activer le nombre d'occurrences et l'on tape 6 dans la zone de texte intitulée Nombre d'occurrences. On clique enfin sur OK pour fermer cette boîte de dialogue.

Lançons une nouvelle session de débogage à son début (on termine la session courante en cliquant le bouton Terminer ou en sélectionnant l'option de menu Exécuter → Terminer). L'exécution sera alors interrompue au sixième passage sur le point d'arrêt (voir figure 3-16). On notera que `value` vaut 1 (voir la vue Expressions), ainsi que l'empilement des appels successifs de `factorial` dans la vue Débogage.

Pour examiner un contexte d'exécution dans la vue Débogage, ainsi que le contenu de ses variables locales, il suffit de cliquer sur un élément de la pile pour le rendre actif. L'exemple de `factorial` engendre un bon nombre de contextes d'exécution – en cliquant sur l'un d'entre eux, on peut voir que toutes les variables locales, comme `value`, ont été préservées dans chacun d'eux. Dans le contexte courant, `value` = 1, dans le précédent `value` = 2, etc.

On vient d'atteindre directement la dernière itération de la méthode `factorial` au moyen d'un compteur d'occurrences. Jusqu'à présent, tout semble correct. Exécutons maintenant le code pas à pas jusqu'à son terme. En cliquant le bouton Avancer d'un pas avec entrée, on entre à nouveau, de façon inattendue, dans `factorial` ; la vue Expressions permet d'observer que la variable `value` vaut 0 : le problème est là, car dans ce cas la méthode retourne 0, ce qui est incorrect. En effet, la factorielle de 0 vaut 1, par définition (et non 0). Dans la situation actuelle, la valeur 0 retournée par la méthode `factorial` est multipliée par les résultats des appels précédents et donne pour résultat final 0.

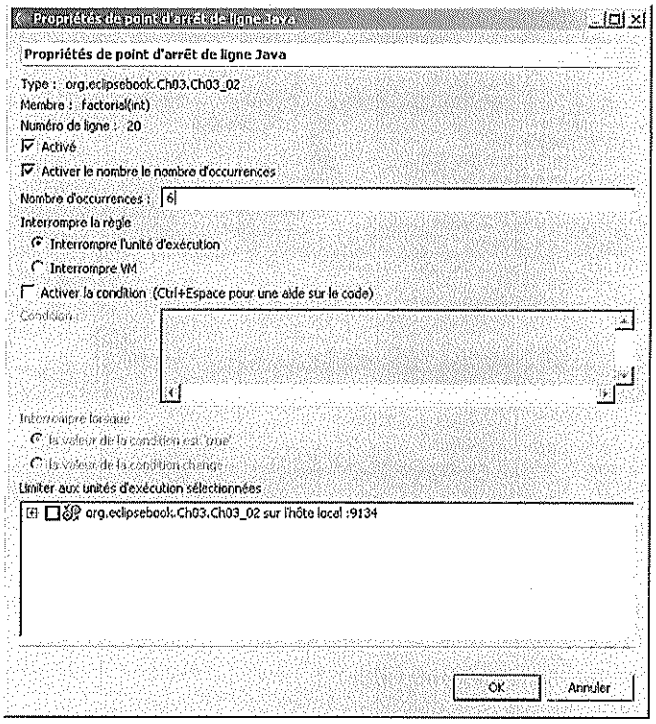


Figure 3-15. Création d'un compteur d'occurrences pour un point d'arrêt

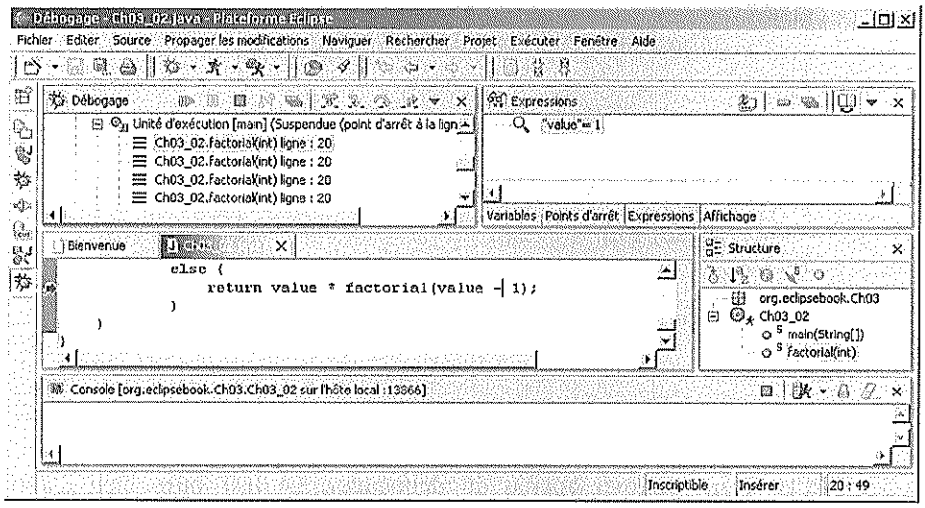


Figure 3-16. Utilisation d'un compteur d'occurrence de point d'arrêt

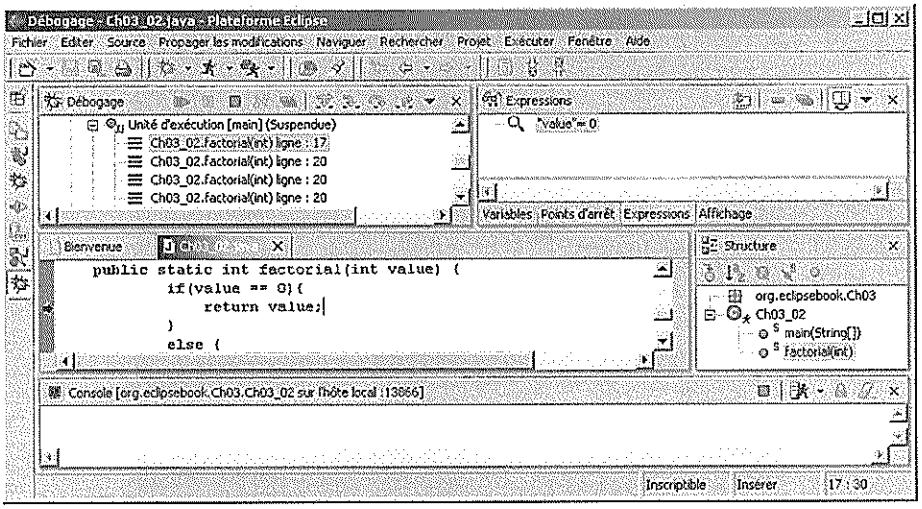


Figure 3-17. Identification de l'erreur

Inspectant notre code, on voit immédiatement qu'il ne faut pas retourner 0 lorsque value est nul (voir la ligne mise en évidence dans la figure 3-17) :

```
public static int factorial(int value) {
        if(value == 0){
                return value;
        }
        else {
                return value * factorial(value - 1);
        }
}
```

On résoud le problème très simplement comme suit :

```
public static int factorial(int value) {
        if(value == 0){
                return 1 ;
        }
        else {
                return value * factorial(value - 1);
        }
}
```

On peut observer le résultat proposé par le code ainsi modifié dans la figure 3-18 : le calcul de la factorielle de 6 donne bien 720.

On vient d'avoir un premier aperçu du débogueur. Le JDT offre bien d'autres possiblités que l'on verra plus loin.

Configuration de points d'arrêts

On a utilisé avantageusement un compteur d'occurrences de points d'arrêt. Un point d'arrêt peut être configuré de nombreuses façons ; pour le constater, il suffit de cliquer le bouton droit de la souris sur un point d'arrêt, puis de sélectionner l'option

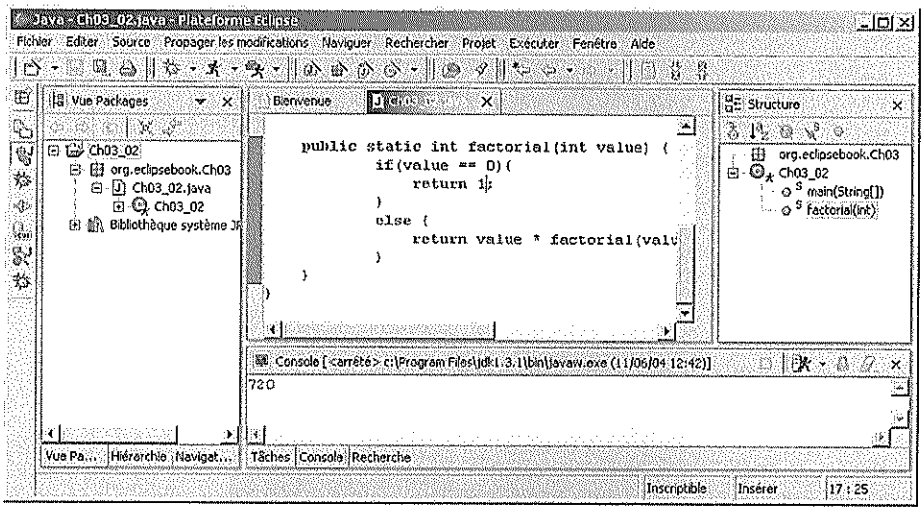


Figure 3-18. Le programme débogué

Propriétés, ce qui ouvre la boîte de dialogue Propriétés de point d'arrêt de ligne Java (voir figure 3-19). En cochant la case à cocher Activer la condition, il est possible de taper une condition (zone de texte intitulée Condition) qui déclenchera l'interruption du programme sur le point d'arrêt. Par exemple, un programme peut poser problème à chaque fois qu'une variable nommée `inventory` contient la valeur 0 ; dans ce cas, on peut taper la condition `inventory == 0` et sélectionner le bouton radio intitulé « la valeur de la condition est 'true' ».

Vérification d'une condition

On peut aussi interrompre l'exécution d'un programme lorsqu'une variable change de valeur – par exemple, si quelque part dans un code, une variable nommée `ipAddress` est modifiée alors qu'elle ne devrait pas l'être, on peut taper le nom de cette variable dans la zone de texte Condition et sélectionner le bouton radio intitulé « la valeur de la condition change » (on peut bien entendu taper toute expression valide dans la zone Condition et pas seulement des noms de variables).

Débogage multi-threads

On peut aussi effectuer des sessions de débogage multi-threads – il est possible de restreindre l'action d'un point d'arrêt à certains threads au moyen de l'option Limiter aux unités d'exécution sélectionnées, ou encore de piloter l'interruption de l'un d'entre eux grâce à Interrompre l'unité d'exécution (qui interrompt l'exécution d'un seul thread d'une application) ou Interrompre VM (qui arrête la machine virtuelle lancée par Eclipse pour déboguer l'application).

Points de contrôle

Les points d'arrêt standard utilisés dans l'exemple précédent sont appelés *points d'arrêt de ligne de code* ; le JDT propose en outre un autre type de point d'arrêt – pour les membres (*points de contrôle*), pour les méthodes et pour les exceptions.

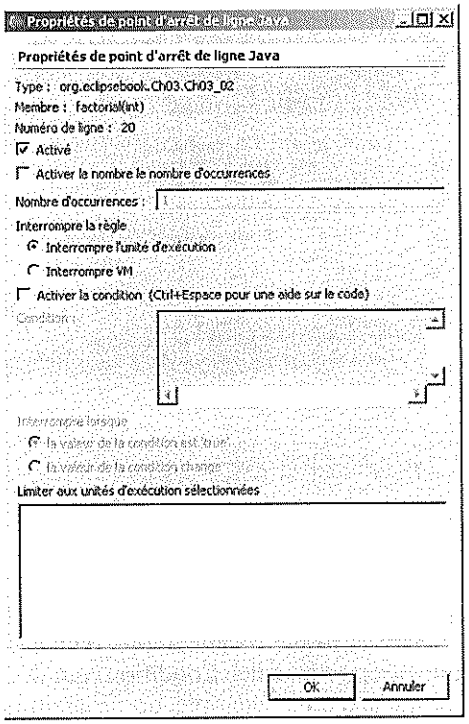


Figure 3-19. Propriétés de points d'arrêt

Les points d'arrêt pour des membres, encore appelés points de contrôle, suspendent l'exécution d'un code lorsque celui-ci est sur le point d'accéder à une donnée membre (en lecture ou en écriture). Les points de contrôle ne peuvent pas être créés sur des variables locales, déclarées dans une méthode : ils ne s'appliquent qu'aux données membres de classes ou d'instances de classes. Ces points de contrôle évitent d'avoir à chercher dans tout le code les instructions procédant à un accès à un membre donné dont on surveille les modifications.

Pour créer un point de contrôle, il suffit de sélectionner un membre dans la vue Java et de cliquer l'option Exécuter → Ajouter/Supprimer un point de contrôle. Le nouveau point de contrôle apparaît dans la vue Points d'arrêt ; on peut le configurer en cliquant dessus avec le bouton droit de la souris et en sélectionnant l'option Propriétés, ce qui ouvre la boîte de dialogue Propriétés de point de contrôle Java de la figure 3-20. On notera en particulier la présence de deux cases à cocher – Accéder et Modification – qui permettent d'activer le point d'arrêt lors d'un accès en lecture et/ou en écriture. On peut aussi utiliser un compteur d'occurrences de la même façon qu'avec un point d'arrêt standard.

Points d'arrêt de méthode

Les points d'arrêt de méthode suspendent l'exécution lorsqu'elle entre ou sort d'un code (suivant la configuration adoptée). Ces points d'arrêt sont utilisés pour des méthodes dont le programmeur ne possède pas le code source. Pour créer un point

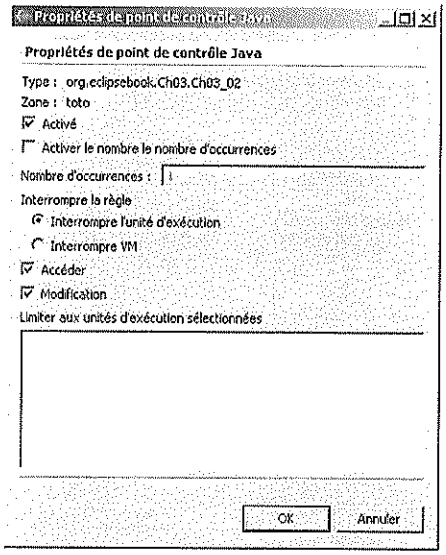


Figure 3-20. Configuration d'un point de contrôle

d'arrêt de méthode, il suffit de sélectionner la méthode concernée dans la vue Java et de cliquer l'option de menu Exécuter → Ajouter/Supprimer un point d'arrêt dans une méthode. On peut configurer ce point d'arrêt en cliquant le bouton droit de la souris sur celui-ci (dans la vue Points d'arrêt) et en ouvrant la boîte de dialogue Propriétés de point d'arrêt de méthode Java de la figure 3-21. On peut préciser si l'exécution du code doit être suspendue lorsqu'on entre et/ou lorsqu'on sort de la méthode, grâce à deux cases à cocher de cette boîte dialogue.

Points d'arrêt d'exception

Eclipse propose aussi des points d'arrêt sur des exceptions, qui suspendent l'exécution d'un code lorsqu'une exception est levée. Cette possibilité est très utile pour localiser l'origine d'une exception (par exemple, `NullPointerException`). On peut aussi suspendre l'exécution du code lorsque l'exception est interceptée (ou non interceptée).

Pour créer un point d'arrêt sur une exception, il faut sélectionner l'option de menu Exécuter → Ajouter un point d'arrêt d'exception Java, qui ouvre la boîte de dialogue de la figure 3-22 ; cette dernière permet de sélectionner le point d'arrêt idoine, et de préciser si l'interruption d'exécution doit se produire lorsque l'exception est interceptée et/ou non interceptée.

On peut configurer les propriétés d'un point d'arrêt sur une exception de la même façon que pour tout autre point d'arrêt : il suffit de cliquer le bouton droit de la souris sur celui-ci, dans la vue Point d'arrêt et de sélectionner l'option Propriétés. Dans la figure 3-23, on peut voir comment on a configuré un point d'arrêt sur les exceptions non interceptées de type `java.lang.NullPointerException`. On peut restreindre l'action du point d'arrêt à certaines portions de code ou utiliser un compteur d'occurrences mais en revanche, et contrairement au points d'arrêt de ligne de code, il n'est pas possible de lui associer une condition de déclenchement.

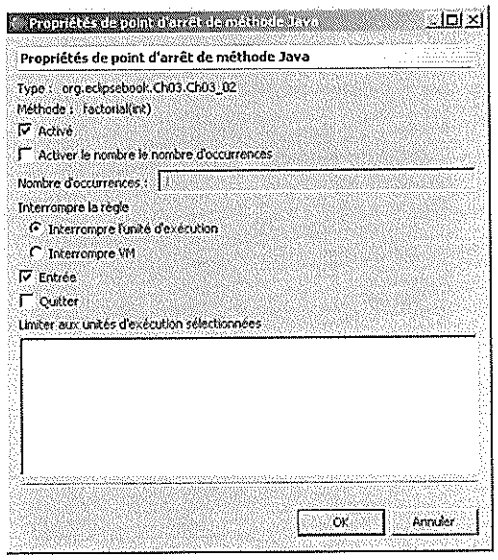


Figure 3-21. Configuration d'un point d'arrêt de méthode

Figure 3-22. Configuration d'un point d'arrêt sur une exception

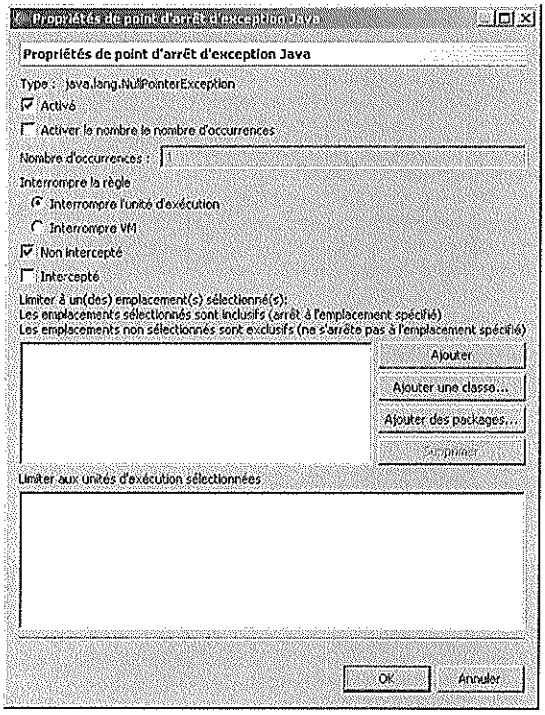


Figure 3-23. Configuration d'un point d'arrêt sur une exception Java

Évaluation d'expressions Java et modification de valeur

Durant une session de débogage, le programmeur peut saisir une expression dans la page détaillée de la vue Expressions (localisée juste au dessus de la barre de défilement horizontale inférieure de cette vue), puis la sélectionner et avec un clic droit de la souris choisir l'option Inspecter. Par exemple, on imagine que la variable `value` a pour valeur 6 ; il est possible de demander l'évaluation de `value + 1` dans la page détaillée en tapant cette expression, puis en la sélectionnant et enfin en choisissant Inspecter dans le menu contextuel. L'expression `value + 1` est alors ajoutée à la liste des expressions figurant dans la vue Expressions et sa valeur affichée (7, en l'occurrence) dans la page détaillée (voir figure 3-24).

On peut aussi éditer la valeur d'un membre ou d'une variable locale en cours de débogage, comme on l'a vu plus haut – il suffit de double-cliquer sur le nom de ce membre ou de cette variable (dans la vue Variables), ce qui ouvre la boîte de dialogue de la figure 3-25. Dans le cas présent, on a double-cliqué sur la variable `value` dans vue Variables ; pour la modifier il suffit de taper la nouvelle valeur et de cliquer sur OK. Cette possiblité est très utile car elle permet d'observer l'impact de différentes valeurs possibles sur l'exécution d'un code – on peut vérifier ce qui se passe lorsqu'une variable nommée à juste titre `denominator` a pour valeur 0 –ou pour tester les réactions d'un code face à des valeurs notoirement pathologiques.

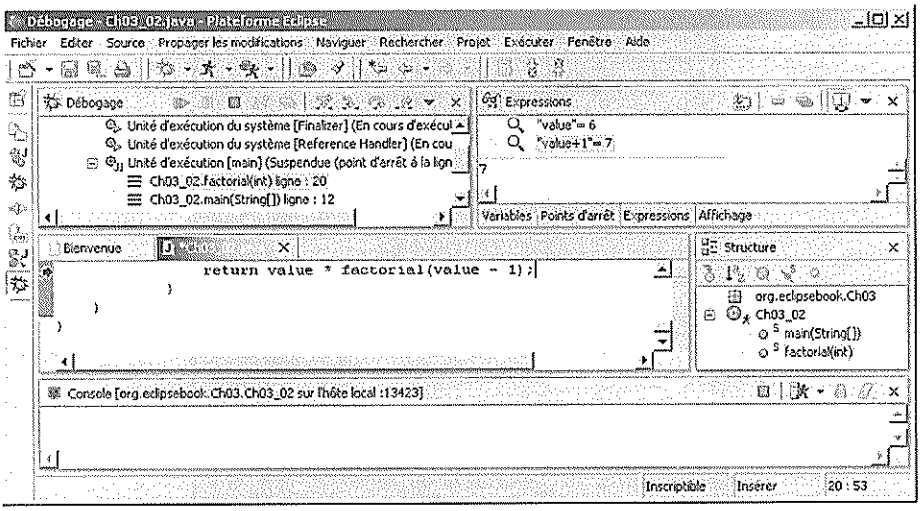


Figure 3-24. Évaluation d'une expression

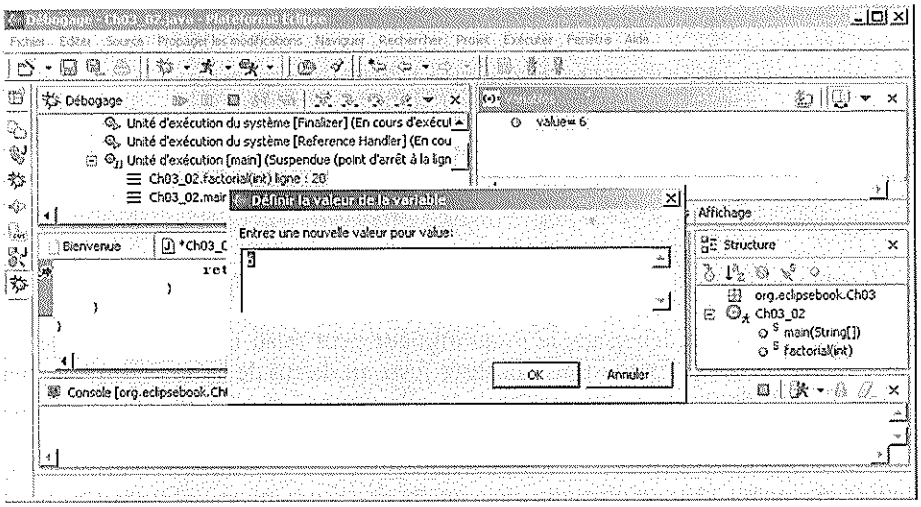


Figure 3-25. Édition de la valeur d'une variable

Modification de code durant une session de débogage

Il est même possible d'éditer du code en cours de débogage sous JDT. Cette posiblité, encore nommée *modification de code durant une session de débogage*, requiert au minimum la version 1.4 du JRE.

L'utilisation la plus simple de la modification de code durant une session de débogage est d'autoriser la compilation automatique du code à chaque fois qu'il est sauvegardé (ce qui n'est pas forcément nécessaire). Pour ce faire, il suffit de cocher l'option Fenêtre → Préférences → Plan de travail → Effectuer une compilation automatique lorsqu'une ressource est modifiée et de cliquer sur le bouton OK.

Le programmeur peut aussi éditer du code durant une session de débogage. Après la modification de celui-ci, il suffit de le sauvegarder et de reprendre l'exécution du programme (si la compilation automatique n'est pas activée, on peut procéder à la compilation de l'application au moyen de l'option Projet → Recompiler tout).

Parfois, on peut aboutir à des résultats étranges lorsque du code est édité en cours de débogage : dans ce cas, il est nécessaire de relancer l'exécution du code à partir de son début.

Éditons par exemple le code de la factorielle lors d'une session de débogage (comme on peut le voir sur la ligne mise en surbrillance, dans la figure 3-26), en modifiant la ligne suivante dans la méthode `factorial` :

```
public static int factorial(int value) {
        if(value == 0){
                return 1;
        }
        else {
                return value * factorial(value - 1);
        }
}
```

Voici la nouvelle ligne :

```
public static int factorial(int value) {
        if(value == 0){
                return 1;
        }
        else {
                return 2 * value * factorial(value - 1);
        }
}
```

Sauvons maintenant le code ainsi modifié, qui va être automatiquement reconstruit si l'option ad hoc a été activée – on peut alors continuer la session de débogage (voir figure 3-26).

Réalisation d'applications pour des version antérieurs du JRE

Que faut-il faire si l'on veut créer un code utilisable avec toutes les versions antérieures du JRE, par exemple jusqu'à la version 1.2, tout en voulant conserver la possibilité de modifier du code durant une session de débogage, qui requiert la version 1.4 ou postérieure ? Il est possible d'utiliser différents JRE pour exécuter et déboguer du code sous Eclipse. Pour modifier la configuration de lancement du débogueur afin qu'un JRE autre que celui par défaut soit utilisé, il suffit de sélectionner Exécuter → Déboguer, de cliquer l'onglet JRE et de sélectionner celui correspondant à ses besoins.

Ainsi se termine notre exploration des outils de débogage JUnit et JDT qui sont, comme on a pu le voir, très puissants. JUnit permet de créer des jeux de tests standard qui peuvent être appliqués à un code en quelques clics de souris. Le débogage sous Eclipse offre de nombreuses options : points d'arrêt configurables, points de contrôle, points d'arrêts sur des exceptions, évaluation d'expressions, exécution pas à pas, modification de la valeur de variable et de membres, modification de code durant une

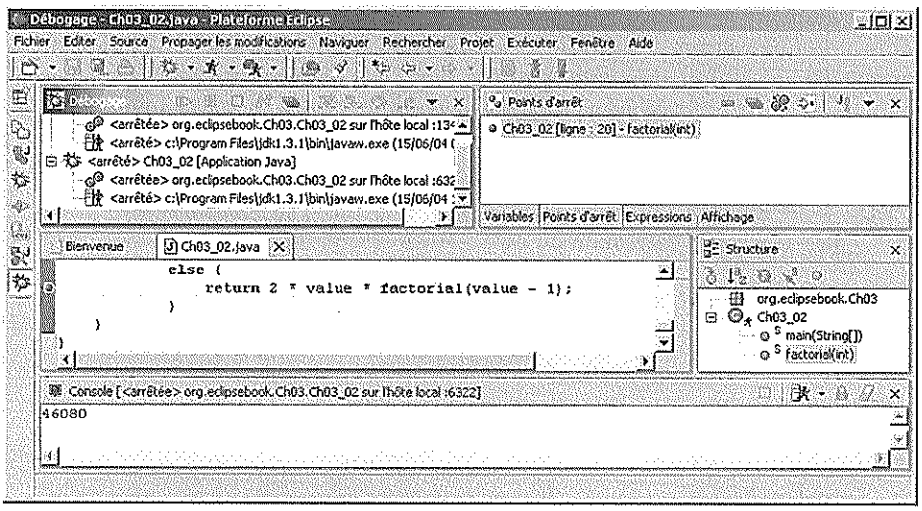


Figure 3-26. Modification de code durant une session de débogage

session de débogage, etc. Toutes ces options vont largement au-delà des possibiltés offertes dans le contexte du développement d'applications sous javac.

CHAPITRE 4

Développement d'applications en équipe sous Eclipse

Les chapitres précédents ont illustré les méthodes développement de code en solo, mais Eclipse est essentiellement destiné à être utilisé en équipe, un cadre de développement plus réaliste dans le contexte professionnel, lié ou non à Java. Sous Eclipse, la gestion du travail en équipe repose sur l'outil CVS (Concurrent Versions System). Le développement d'applications en équipe et la gestion du travail de ses membres à l'aide de CVS constituent l'objet de ce chapitre.

Principes du contrôle du code source

Lorsque des développeurs travaillent en équipe, il est indispensable que leurs différentes tâches soient bien coordonnées. Cette coordination englobe bien sûr tous les aspects liés à la conception, à la réalisation et au planning de travail, car même avec les meilleures intentions du monde, un groupe de programmeurs mal coordonnés peut rapidement connaître des conflits nuisibles au travail du groupe. Par exemple, les corrections de code judicieuses d'un des membres peuvent être malencontreusement effacées par un collègue ignorant les modifications effectuées.

Le contrôle du code source permet d'éviter ce genre de problème. Ce contrôle s'appuie sur des outils de contrôle d'accès au code et de maintien d'un historique des modifications effectuées dans celui-ci, minimisant ainsi les risques d'effacement malencontreux. L'existence d'un historique du code source constitue un outil très puissant – il permet non seulement de comparer une version récente et boguée d'un code à une version plus ancienne, mais aussi de revenir à une version antérieure si les modifications récentes ont débouché sur des catastrophes ingérables.

Le contrôle du code source permet de coordonner le développement simultané de différentes versions d'un logiciel – par exemple, un développeur peut vouloir travailler simultanément sur la version diffusable de son code et sur une nouvelle version bêta. Il est possible de gérer cette activité au moyen de *branches*, comme nous allons le voir dans ce chapitre.

Les principes de fonctionnement de CVS

CVS est un projet open source ; initié en 1986 sous la forme d'un ensemble de scripts sous shell Unix, CVS se transforma en un logiciel dédié en 1989. Actuellement, CVS

est supporté par la plupart des systèmes d'exploitation – Unix, Linux, Windows, etc. Le lecteur intéressé par l'histoire détaillée de CVS pourra consulter le site *http://www.cvshome.org*.

Le principe central de CVS, comme pour beaucoup d'autres référentiels d'archivage, est de gérer et de stocker toutes les modifications effectuées sur un code source. En fait, de nombreux types de référentiels d'archivage sont disponibles, dont certains sont plus puissants que CVS ; néanmoins, CVS est le plus répandu (peut-être parce qu'il est gratuit).

Sous CVS, un *module* est l'équivalent d'un projet Eclipse. Les modules CVS sont représentés sous la forme de dossiers. Un projet standard correspond à un *module physique*, alors qu'un *module logique* ou *virtuel* correspond à un ensemble de ressources liées.

Les fichiers partagés sont stockés dans le *référentiel* CVS. Lorsque l'un d'entre eux est ôté du *référentiel* par le programmeur, ce fichier est *réservé*. Lorsqu'un fichier est modifié, le programmeur *valide* ces modifications pour enregistrer et transmettre celles-ci au référentiel. Pour obtenir sa copie personnelle et à jour d'un fichier géré sous CVS, le programmeur procède à une *mise à jour* de celui-ci à partir du référentiel.

On distingue en général deux modèles de référentiel pour le code source :

Verrouillage pessimiste
: Un fichier donné ne peut être réservé que par un seul programmeur à la fois – lorsqu'un fichier est réservé, celui-ci est verrouillé, au sens où les autres programmeurs ne peuvent récupérer que des copies en lecture seule du fichier : ils ne peuvent en aucun cas modifier ces copies. L'accès aux fichiers est séquentiel.

Verrouillage optimiste
: Les fichiers peuvent être réservés et modifiés librement par tous les programmeurs. Lorsqu'un fichier est validé, le logiciel de gestion du référentiel *fusionne* les modifications effectuées automatiquement. Si cette opération ne peut être menée à bien par le logiciel, celui-ci notifie au programmeur qu'il lui faut faire ce travail lui-même. L'accès au fichier est libre.

Par défault, CVS suit les règles du verrouillage optimiste – bien que certains logiciels CVS suivent les règles du verrouillage pessimiste. Dans la suite de cet ouvrage, on suivra celles du verrouillage optimiste, qui sont aussi les règles d'Eclipse.

Comme il est nécessaire d'avoir une traçabilité complète pour chaque fichier, CVS attribue automatiquement un numéro de version à chaque fichier. À chaque fois qu'un fichier est validé, son numéro de version est incrémenté. La première version d'un fichier a pour numéro de version 1.1 ; son numéro de version suivant est 1.2, etc. (cette numérotation peut dépendre du serveur CVS ; certains débutent celle-ci à 1.0). Lorsqu'un fichier est validé dans le référentiel, celui-ci reçoit un nouveau numéro de version automatiquement. On peut l'observer dans les différentes vues offertes au programmeur par Eclipse lorsqu'on utilise le contrôle du code source.

Lorsqu'une mise à jour d'un fichier est effectuée à partir du référentiel, la version locale de ce fichier n'est pas écrasée. Eclipse fusionne la version locale avec celle extraite du référentiel. S'il y a un conflit, les lignes de code à l'origine de celui-ci sont marquées au moyen d'une balise CVS spéciale indiquant les problèmes potentiels. Le programmeur a la possibilité de modifier son code afin de résoudre les conflits issus

de la fusion ; une fois ce travail effectué, le programmeur peut valider une nouvelle version de ce code. En général, les mises à jour sont réalisées sans problème, en particulier si le programmeur travaille seul sur un sous-ensemble précis du projet. Cependant, une mise à jour peut parfois soulever des problèmes délicats qu'il faut résoudre à la main.

CVS supporte aussi les *branches* de développement. Le fil principal de développement, encore appelé tête, est identifié par la marque spéciale `HEAD`. Les branches constituent des bifurcations à partir de la tête ; dans une branche, les modifications de code sont effectuées indépendamment du travail réalisé dans le fil principal de développement. Par exemple, c'est ce qui se passe lorsqu'on réalise une version bêta d'un projet en marge du travail principal de développement. Le programmeur doit alors attribuer une marque à cette branche ; cette marque permettra à CVS de garder toutes les traces des modifications effectuées dans celle-ci. Nous allons examiner la mise en place de tous ces outils de gestion dans la suite de ce chapitre.

Recherche d'un serveur CVS

Pour travailler avec CVS, il est nécessaire d'avoir accès à un serveur CVS. Comment obtenir un tel accès ?

La plupart des configurations Linux et Unix sont installées avec un serveur CVS intégré. Pour tester la disponiblité de celui-ci, il suffit de taper `cvs --help` en ligne de commande, ce qui permet d'obtenir la liste des différentes options proposées par l'aide. Un programmeur ne disposant pas d'un serveur CVS peut en télécharger un à partir du site *http://www.cvshome.org*.

Sous Windows, le problème est un peu plus complexe. Il existe de nombreux serveurs CVS pour Windows, tels que CVSNT, disponibles gratuitement sur le site *http://www.cvsnt.org*. Pour installer CVSNT, il suffit de télécharger l'exécutable et de le lancer.

Création d'un référentiel

Il faut ensuite créer un référentiel pour le code source du projet au moyen du serveur CVS. Sous Linux ou sous Unix, il suffit pour cela de taper la commande `cvs -d path init`, où *path* désigne la localisation du répertoire qui doit être utilisé comme référentiel (les privilèges liés à *path* doivent être initialisés de la même façon pour tous les membres de l'équipe de développement, afin que ceux-ci puissent accéder au référentiel).

Sous CVSNT, il faut sélectionner l'onglet Repositories (référentiels) de la fenêtre de contrôle de CVSNT (service control panel), puis cliquer le bouton Add (ajouter), taper le nom complet du répertoire constituant le référentiel (par exemple *c:\referentiel*) et cliquer sur le bouton OK.

Connexion à CVS

Sous Linux et sous Unix, deux options d'accès à CVS sont possibles : SSH (secure shell) et pserver. On utilisera la seconde (pserver) dans cet ouvrage, mais les deux sont utilisables, à condition de s'assurer que le protocole correct fonctionne bien sur la plateforme utilisée.

Sous Windows, CVSNT se comporte comme un service Windows : il est accessible à Eclipse dès lors qu'il est en fonctionnement. Pour lancer CVSNT, il suffit de sélectionner le « service control panel » dans le menu Démarrer, et de cliquer le bouton Start dans les deux zones intitulées « CVS Service » et « CVS Lock Service » : le message « Running » (fonctionnement en cours) apparaît alors dans ces deux zones, comme on peut le voir dans la figure 4-1.

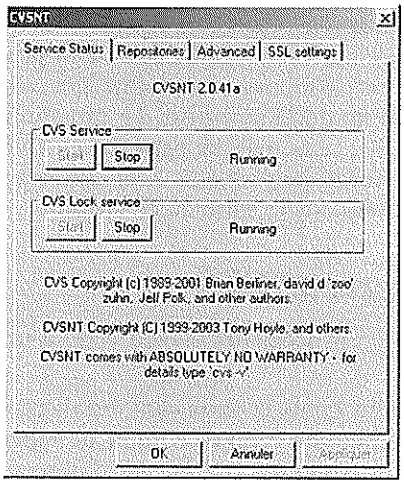


Figure 4-1. Démarrage de CVSNT

CVS fonctionne : on peut maintenant l'utiliser sous Eclipse.

Ajout d'un projet dans le référentiel CVS

Utilisons CVS en ajoutant un nouveau projet, `Ch04_01`, dans le référentiel CVS. Une fois ce projet ajouté, tout programmeur ayant accès au référentiel pourra le réserver et travailler sur celui-ci. Le projet `Ch04_01` est présenté dans l'Example 4-1 ; le code de ce projet se contente d'afficher le message « Bonjour chez vous ».

Exemple 4-1. A sample project

```
package org.eclipsebook.ch04;

public class Ch04_01 {

        public static void main(String[] args) {
                System.out.println("Bonjour chez vous");
        }
}
```

Création d'un emplacement du référentiel

Comment faire pour ajouter un projet au référentiel CVS, une fois ce projet créé ? Il faut tout d'abord faire connaître à Eclipse quelques informations relatives au

référentiel ; pour cela, il faut cliquer l'option de menu Fenêtre → Ouvrir la perspective → Autre... et sélectionner Exploration du référentiel CVS (une fois cette opération effectuée, Eclipse ajoute un bouton raccourci sur cette perspective dans la barre verticale sur trouvant sur la gauche de la fenêtre d'Eclipse). Dans cette perspective, on peut cliquer le bouton droit de la souris sur la vue Référentiels CVS (actuellement vide de tout contenu) et sélectionner Nouveau → Emplacement du référentiel : la boîte de dialogue Ajout d'un référentiel CVS apparaît (voir figure 4-2).

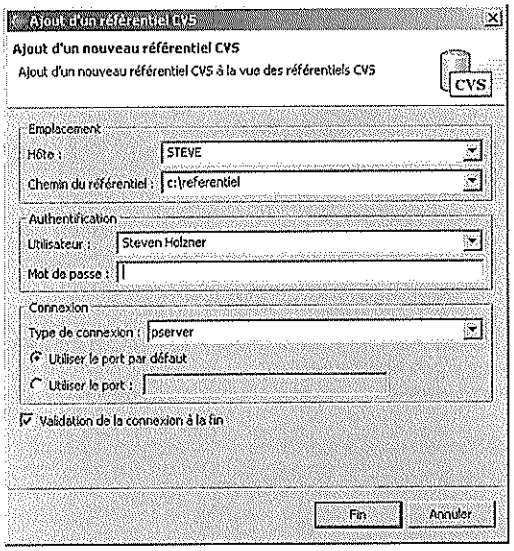


Figure 4-2. Connexion d'un référentiel CVS à Eclipse

Dans cette boîte, il faut taper le nom du serveur hôte (en général, c'est le nom de l'ordinateur sur lequel se trouve le serveur), le nom du répertoire servant de référentiel, le nom de l'utilisateur et son mot de passe, et enfin le type de connexion (on utilisera ici le protocole pserver). On clique alors le bouton Fin pour ajouter le nouveau référentiel à la vue Référentiels CVS (voir figure 4-3).

Nous voici connectés à un référentiel CVS. L'étape suivante consiste à mettre en place le partage du nouveau projet.

Il existe même un serveur public dédié à Eclipse. Il est possible de créer un référentiel sur ce serveur, à l'adresse `:pserver:anonymous@dev.eclipse.org:/home/eclipse`.

Partage de projet

Pour ajouter le projet `Ch04_01` au référentiel CVS, il suffit d'utiliser les fonctionnalités CVS offertes par Eclipse. Pour cela, il faut ouvrir la perspective Java, cliquer le bouton droit sur le projet à partager, puis sélectionner l'option Equipe → Partager le projet. La boîte de dialogue Partager le projet avec le référentiel CVS apparaît alors (voir figure 4-4).

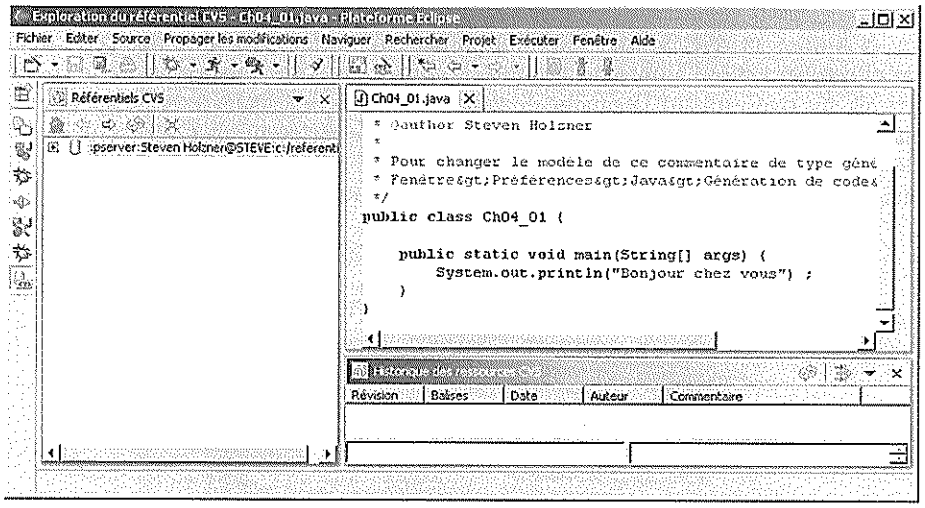


Figure 4-3. Un nouveau référentiel dans la vue Référentiels CVS

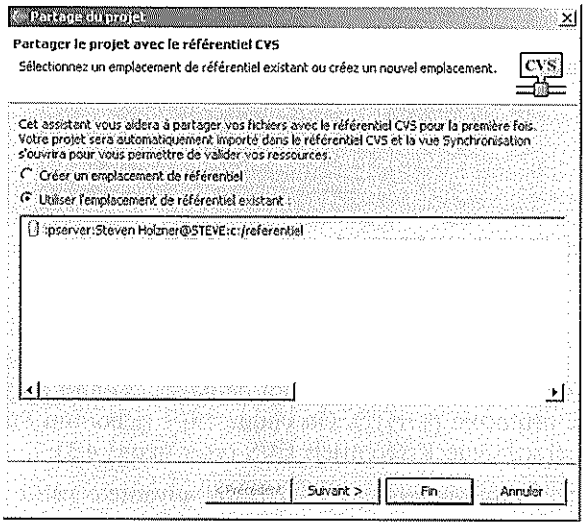


Figure 4-4. Identification du référentiel à utiliser

Dans cette boîte de dialogue, il convient de s'assurer que l'option « Utiliser l'emplacement de référentiel existant » est sélectionnée. On clique alors sur le référentiel créé précédemment, puis sur le bouton Fin. Le projet est alors ajouté au référentiel.

Cette opération donne au module CVS le même nom que celui du projet. Pour donner à un module un nom différent de celui du projet, il suffit de cliquer le bouton Suivant, de taper le nom de ce module puis de cliquer sur le bouton Fin.

Les opérations ci-dessus ajoutent le projet au référentiel et ouvrent la vue Synchronisation. Cette vue permet de synchroniser les fichiers, mais elle n'est pas très

utile pour le moment ; on verra son intérêt plus loin dans ce chapitre, pour synchroniser la version locale du code lorsque le code détenu par le référentiel a été modifié par un autre membre de l'équipe.

Le programmeur peut voir les commandes envoyées par Eclipse au serveur CVS en ouvrant la console CVS au moyen de l'option Fenêtre → Afficher la vue → Autre → CVS → Console CVS. La console CVS apparaît alors en se superposant à la vue Console.

Validation de fichier

Jusqu'à présent, on a simplement ajouté le projet `Ch04_01` au référentiel CVS ; nous allons maintenant enregistrer le fichier de code auprès du référentiel. Cette opération est réalisée en deux étapes : on *ajoute* tout d'abord le fichier au référentiel CVS (cette opération n'a d'autre but que de notifier à CVS l'existence du fichier, et pas de procéder à la sauvegarde du fichier lui-même) ; on *valide* ensuite celui-ci (ce qui insère effectivement le fichier dans le référentiel et le rend partageable).

On peut ajouter un fichier individuellement dans le référentiel en cliquant le bouton droit de la souris sur ce fichier et en sélectionnant l'option Equipe → Ajouter au contrôle de version. On sélectionne ensuite l'option de menu Equipe → Valider pour valider ce fichier.

Il existe cependant une méthode plus simple. Eclipse fournit un raccourci à cet effet – il suffit de cliquer le bouton droit de la souris sur le projet, dans la Vue Packages, et de sélectionner Equipe → Valider. En conséquence Eclipse affiche la boîte de dialogue Ajout au contrôle de version CVS, qui liste les fichiers non encore ajoutés dans le référentiel CVS. On peut sélectionner tout les fichiers en cochant la case à cocher intitulée Sélectionner tout, puis en cliquant sur le bouton Oui. Eclipse propose alors d'éditer un commentaire de validation concernant les nouveaux fichiers à valider, ce qui permet au programmeur d'étiqueter l'ensemble de ces fichiers et de souligner leurs spécificités par rapport aux autres fichiers intégrés dans le référentiel. Dans ce cas, on peut simplement taper un texte tel que `Build 1.0` et cliquer OK.

Pour enregistrer ou réserver (auprès du référentiel) des projets Eclipse, il convient de valider le fichier *.project* lui-même.

Pour vérifier que le projet est bien connecté au référentiel CVS, on peut sélectionner Fenêtre → Préférences → Plan de travail → Décorations d'intitulés, cocher la case à cocher CVS et cliquer OK. Cela affiche un cylindre doré à côté des fichiers intégrés dans le référentiel CVS (voir figure 4-5). Ces fichiers sont assortis d'un numéro de version ; dans le cas présent, il s'agit de la version 1.1.

Basculons sur la perspective Exploration du référentiel CVS (voir figure 4-6). On peut y voir l'intégralité du projet, matérialisé par le module CVS `Ch04_01`, dans la section `HEAD` du référentiel, qui constitue le fil principal de développement. On remarque aussi la présence du répertoire *CVSROOT*, qui contient les données relatives à l'administration du système CVS et les nœuds de branches, qui contiennent tous les fichiers des autres branches de développement.

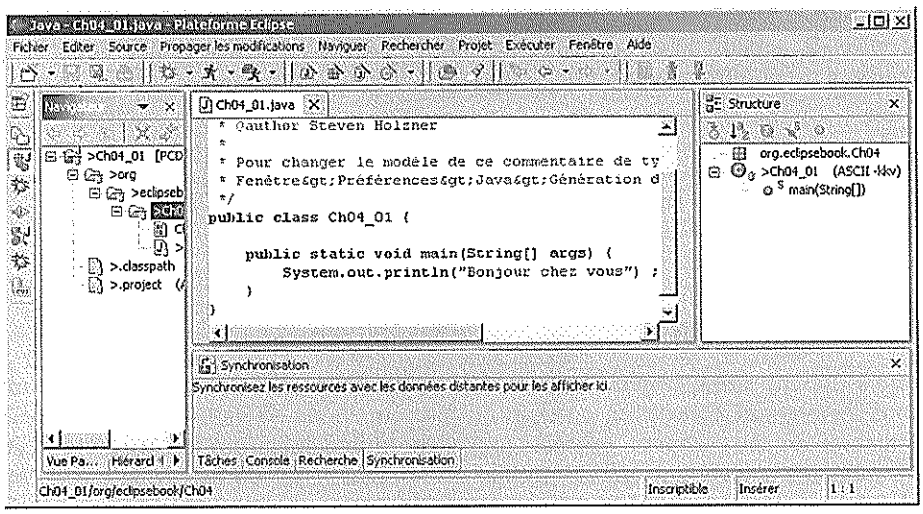


Figure 4-5. Vérification de la connexion à CVS

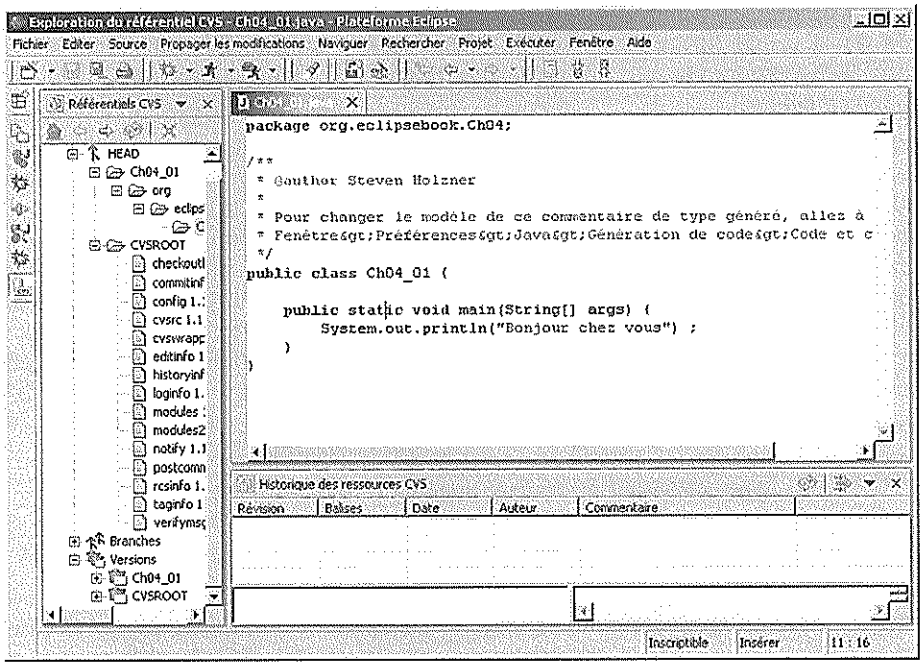


Figure 4-6. Examen des fichiers du référentiel CVS

Réservation de projet

Comment un collaborateur peut-il réserver le nouveau module créé dans ce chapitre ? Il lui faut tout d'abord se connecter au référentiel comme on l'a fait plus haut – en cliquant le bouton droit de la souris dans la vue Référentiels CVS, en sélectionnant

Nouveau → Emplacement du référentiel et en tapant l'emplacement du référentiel, le nom de l'utilisateur, son mot de passe et le type de connexion.

Le programmeur peut ouvrir la vue Référentiels CVS pour explorer les fichiers contenus dans celui-ci. Pour réserver le module `Ch04_01`, il suffit de cliquer dessus dans la vue Référentiels CVS, et avec le bouton droit de la souris de sélectionner l'option du menu contextuel Réserver en tant que, ce qui ouvre la boîte de dialogue Réservation de la figure 4-7.

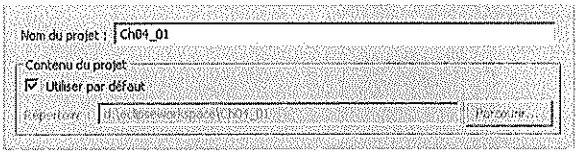


Figure 4-7. Réservation d'un projet

Dans le cas présent, un clic sur le bouton OK déclenche la réservation du projet `Ch04_01` ainsi qu'une tentative de compilation de celui-ci. Si l'on partage un projet Eclipse et si chaque module CVS possède son propre fichier Eclipse *.project*, il suffit de sélectionner l'option Réserver en tant que projet du menu contextuel, de la vue Référentiels CVS, ce qui provoque la réservation d'un projet Eclipse ainsi qu'une tentative de compilation locale. Si le code concerné n'est pas dans un projet compatible avec Eclipse, il sera demandé au programmeur le type de projet à créer ; il suffit alors de sélectionner l'option Java, puis l'option Projet Java.

Mise à jour de code

Que se passe-t-il si plusieurs membres de l'équipe effectuent des modifications sur les fichiers du projet `Ch04_01` et s'ils les sauvegardent ? Imaginons par exemple que l'un des membres modifie le code ci-dessous (qui affiche le message « Bonjour chez vous ») :

```
public class Ch04_01 {

        public static void main(String[] args) {
                System.out.println("Bonjour chez vous");
        }
}
```

par le code ci-dessous (qui affiche le message « Salut à tous ») :

```
public class Ch04_01 {

        public static void main(String[] args) {
                System.out.println("Salut à tous");
        }
}
```

Une fois cette modification effectuée et le fichier *Ch04_01.java* sauvegardé, on voit apparaître un > en face de divers fichiers de la Vue Packages (voir figure 4-8) ; ces symboles indiquent l'existence de modifications de code non encore validées par le programmeur.

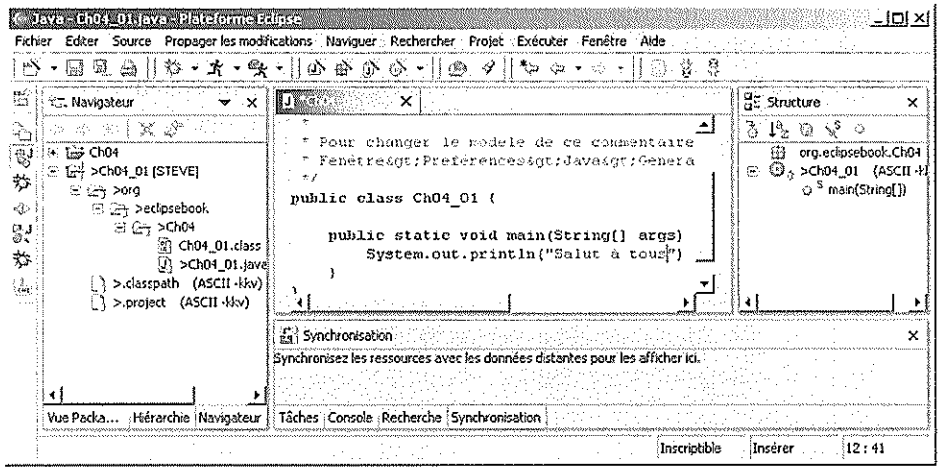


Figure 4-8. Modification de fichier dans le référentiel CVS

On pourrait dès maintenant valider ces modifications de code dans le référentiel CVS, mais, en règle générale, il vaut mieux ne pas valider ainsi sans savoir au préalable comment elles seront traitées par le référentiel. Imaginons par exemple qu'un autre membre de l'équipe ait extrait du référentiel CVS le code qui affiche « Bonjour chez vous » et qu'il ait modifié le message à afficher en « Salut la compagnie » :

```
public class Ch04_01 {

        public static void main(String[] args) {
                System.out.println("Salut la compagnie");
        }
}
```

Cet autre membre de l'équipe a validé ses modifications en cliquant le bouton droit de la souris sur le fichier concerné, sous sa propre plateforme Eclipse et en sélectionnant l'option Valider. Sa version du fichier est donc devenue la version 1.2. Des modifications de ce type sont très courantes et il convient de s'assurer des modifications éventuellement réalisées par d'autres membres de l'équipe avant de valider ses propres modifications. Pour accéder aux modifications des collègues de l'équipe, il suffit de cliquer le bouton droit de la souris sur le fichier concerné et de sélectionner l'option de menu Equipe → Mettre à jour. Cette action déclenche l'importation de la version 1.2 de *Ch04_01.java* sur sa propre plateforme Eclipse, comme on peut le constater dans la vue Packages, à gauche de la figure 4-9. Dans le cas présent, la vue indique aussi l'existence d'un conflit entre la version locale du code et celle du référentiel, comme on peut le voir dans l'éditeur.

S'il n'y avait pas eu de conflit, les modifications auraient simplement été fusionnées avec la version locale du code. Comme les modifications locales et celles enregistrées dans le référentiel concernent la même ligne de code, un conflit apparaît. L'outil CVS inclus dans Eclipse précise la nature du conflit en listant les deux versions de la même ligne de code, documentées à l'aide de balises CVS :

```
public class Ch04_01 {
```

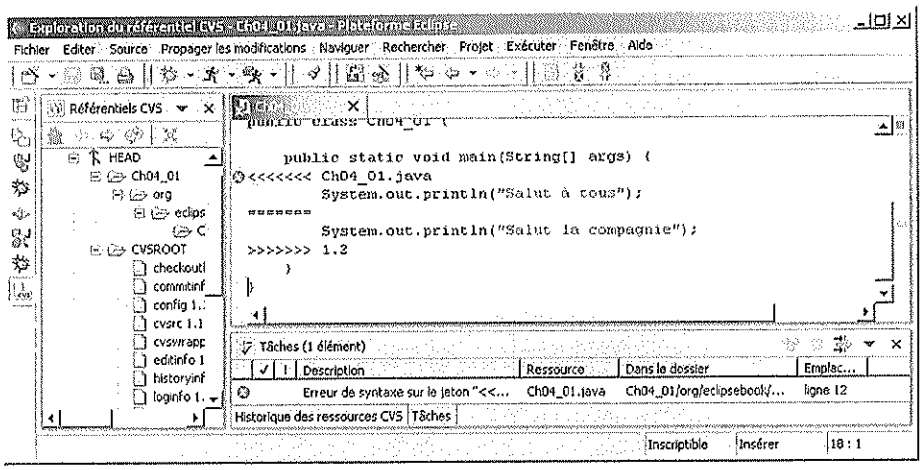


Figure 4-9. Mise à jour d'un fichier modifié

```
        public static void main(String[] args) {
<<<<<<< Ch04_01.java
                System.out.println("Salut à tous");
=======
                System.out.println("Salut la compagnie");
>>>>>>> 1.2
        }
}
```

Pour obtenir plus d'information, il suffit de cliquer le bouton droit de la souris sur le fichier modifié, *Ch04_01.java*, et de sélectionner l'option Equipe → Afficher dans l'historique des ressources. La vue Historique des ressources CVS apparaît alors (voir figure 4-10), qui affiche les commentaires pour chaque version du code de la vue.

Comparaison de code avec l'historique local

Afin de rendre plus évidentes les modifications engendrées par CVS dans de longs fichiers, on peut comparer *Ch04_01.java* avec son historique local. Pour ce faire, il faut cliquer le bouton droit de la souris sur le fichier concerné et sélectionner l'option Comparer avec → Historique local, ce qui ouvre la boîte de dialogue Comparer avec l'historique local de la figure 4-11. Celle-ci met en évidence de façon graphique les modifications effectuées par l'outil CVS intégré dans Eclipse.

Validation de code

Avant de valider un fichier, il appartient au programmeur de résoudre les conflits révélés par la mise à jour du code. Dans le cas présent, nous allons simplement accepter les modifications effectuées par le collègue, mais en ajoutant une modification personnelle – au lieu d'afficher « Salut la compagnie », le code devra afficher « Salut la compagnie et tous ses amis » :

```
public class Ch04_01 {
```

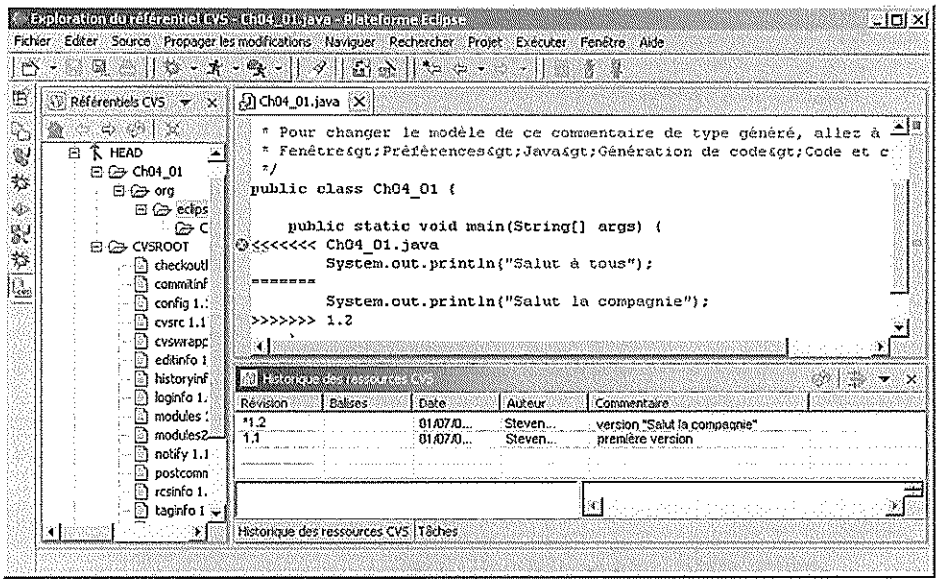


Figure 4-10. Examen de l'historique des ressources

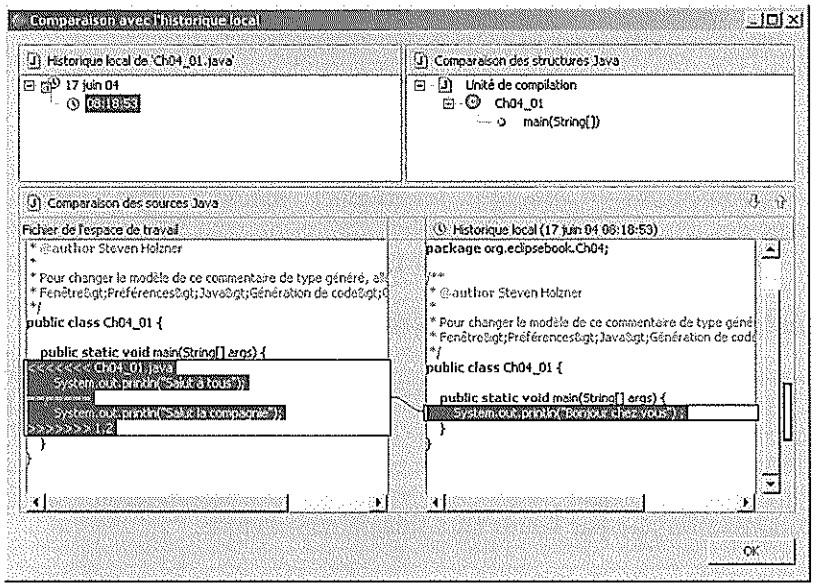


Figure 4-11. Vérification de l'historique local d'un fichier

```
        public static void main(String[] args) {
                System.out.println("Salut la compagnie et tous ses amis");
        }
}
```

À présent, on clique le bouton droit de la souris sur *Ch04_01.java* et on sélectionne Equipe → Valider, ce qui ouvre la boîte de dialogue Valider de la figure 4-12. Cette

boîte de dialogue demande un nouveau commentaire concernant le fichier en cours de validation ; dans le cas présent, le commentaire sera « version "Salut ... ses amis" ».

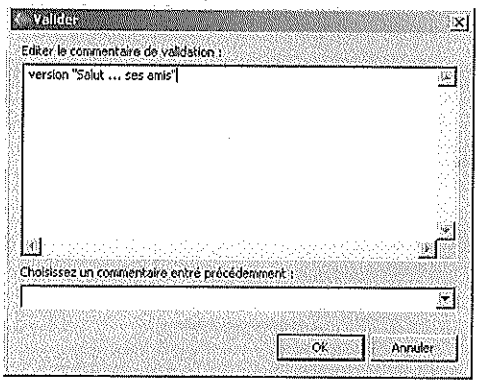


Figure 4-12. Validation d'un fichier

Il faut alors cliquer sur le bouton OK pour valider ces modifications dans le référentiel CVS. La version 1.3 du fichier est alors créée. L'exemple ci-dessus a permis de présenter toutes les étapes clés du cycle de développement et de l'utilisation de CVS : enregistrement du code auprès du référentiel, réservation, mise à jour afin de connaître les modifications effectuées par les autres membres de l'équipe et validation du code. Si les différences entre la version locale et celle du référentiel sont très importantes, on procédera à une *synchronisation* de code.

Synchronisation de code

Lorsque les différences entre la version locale du code et celle du référentiel sont relativement limitées, une mise à jour est parfaitement satisfaisante. En revanche, si les différences sont très substantielles, il est préférable de procéder à une synchronisation de code. Cette technique permet au programmeur de comparer les modifications effectuées par tous les membres du groupe, selon un format plus facile à utiliser que le format fusionné issu de la mise à jour.

Imaginons par exemple qu'un autre membre du groupe travaille sur le même fichier que celui de notre exemple, *Ch04_01.java*, qu'il ajoute une nouvelle ligne de code et qu'il valide son fichier qui devient la version 1.4 du code :

```
public class Ch04_01 {

        public static void main(String[] args) {
                System.out.println("Salut la compagnie");
                System.out.println("La vie est belle ?");
        }
}
```

Pour synchroniser la version locale du code, qui est la version 1.3, avec cette nouvelle version de *Ch04_01.java* (qui constitue la version 1.4 du référentiel), il suffit de cliquer sur le projet `Ch04_01` avec le bouton droit de la souris et de sélectionner Equipe → Synchroniser avec le référentiel. On obtient alors le résultat présenté dans la

figure 4-13. Pour voir apparaître les différences entre la version locale du code et celle du référentiel, dans la vue Comparaison du source Java, il suffit de double-cliquer sur le fichier concerné, dans la vue Comparaison de la structure.

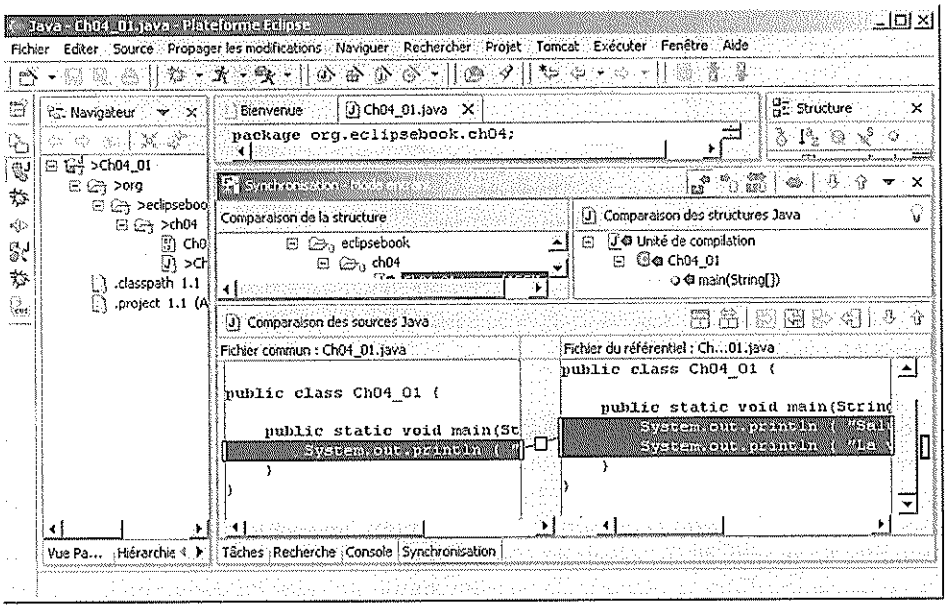


Figure 4-13. Synchronisation d'un projet

Lorsqu'on procède pour la première fois à une synchronisation avec le référentiel, on peut voir que les fichiers ayant fait l'objet de modifications sont marqués d'une double flèche rouge (en forme de diamant dans la figure 4-13) dans la vue Synchronisation. En double-cliquant sur un tel fichier, on obtient la visualisation de la figure 4-13, qui permet de comparer directement la version (1.4) du référentiel à la version (1.3) locale du fichier, dans la vue Comparaison du source Java se trouvant en bas de la fenêtre.

En particulier, on notera la présence d'une ligne reliant deux boîtes de code, dans la vue Comparaison du source Java, ainsi que d'un petit carré blanc au milieu de cette ligne. Si on place le curseur de la souris sur ce carré, celui-ci se transforme en un bouton représentant une flèche pointant à gauche. Si l'on clique cette flèche, on déclenche l'importation des modifications vers la version locale du fichier : cette méthode permet de gérer plus facilement la synchronisation du code. Une fois cette tâche réalisée, on peut alors valider ses propres modifications auprès du référentiel.

Dans le cas présent, le fichier du référentiel a été modifié postérieurement à la dernière synchronisation/mise à jour du code, et la vue Synchronisation s'est ouverte en *mode entrant* (comme on peut le voir dans la barre de titre de cette vue, dans la figure 4-13) ; cette vue montre les modifications faites par rapport au référentiel. La vue Synchronisation propose trois modes :

Entrant

Ce mode permet d'afficher les fichiers qui ont été modifiés dans le référentiel depuis la dernière opération de synchronisation, de mise à jour ou de validation du code local auprès du référentiel.

Sortant

Ce mode permet d'afficher les fichiers modifiés par le programmeur depuis la dernière opération de synchronisation, de mise à jour ou de validation du code local auprès du référentiel effectuée par lui-même.

Entrant/sortant

Ce mode permet d'afficher les fichiers locaux ou ceux du référentiel qui ont été modifiés depuis la dernière opération de synchronisation, de mise à jour ou de validation du code local auprès du référentiel effectuée par le programmeur.

On peut sélectionner le mode de synchronisation au moyen des boutons visibles sur la droite de la barre de titre de la vue Synchronisation. Dans la figure 4-13, Eclipse a détecté des modifications en provenance du référentiel et a donc basculé en mode entrant, ce qui signifie que le bouton « mode entrant » est enfoncé. À droite de ce bouton se trouve respectivement les boutons des modes sortant et entrant/sortant.

Imaginons par exemple que l'on modifie la version locale de *Ch04_01.java* afin que le message affiché ne soit plus « Salut la compagnie et tous ses amis » mais « Salut la compagnie et tous ses amis d'aujourd'hui » :

```
public class Ch04_01 {

   public static void main(String[] args) {
      System.out.println("Salut la compagnie et tous ses amis d'aujourd'hui");
   }
}
```

Une fois cette modification sauvegardée localement et synchronisée, on peut sélectionner le mode sortant, qui permet de voir les différences entre la version locale et la version du référentiel (voir figure 4-14).

Ces modes sont conçus pour permettre de filtrer les modifications effectuées et rendre ainsi la mise à jour et la synchronisation plus faciles à mettre en œuvre. Si beaucoup de modifications sont effectuées, il est préférable de mettre à jour et de synchroniser le code fréquemment – même si elle est un peu contraignante, cette pratique permet de travailler en équipe en évitant que la fusion du code entre les versions locales et la version du référentiel devienne impossible (ce qui arrive s'il s'écoule trop de temps entre deux mises à jour ou deux synchronisations).

Autre détail important : bien que les modifications effectuées dans une version locale d'un code soient réversibles (option de menu Editer → Annuler), cela n'a aucun impact sur la version du référentiel CVS. Si l'on effectue beaucoup de modifications dans la version locale d'un fichier et si l'on constate qu'il y en a trop, au total, par rapport à la version fusionnée avec le référentiel, il est possible d'utiliser l'option Annuler. Cette option n'agira que sur la version locale : lors de la prochaine synchronisation avec la version du référentiel, des options différentes de celles présentées plus haut peuvent être proposées.

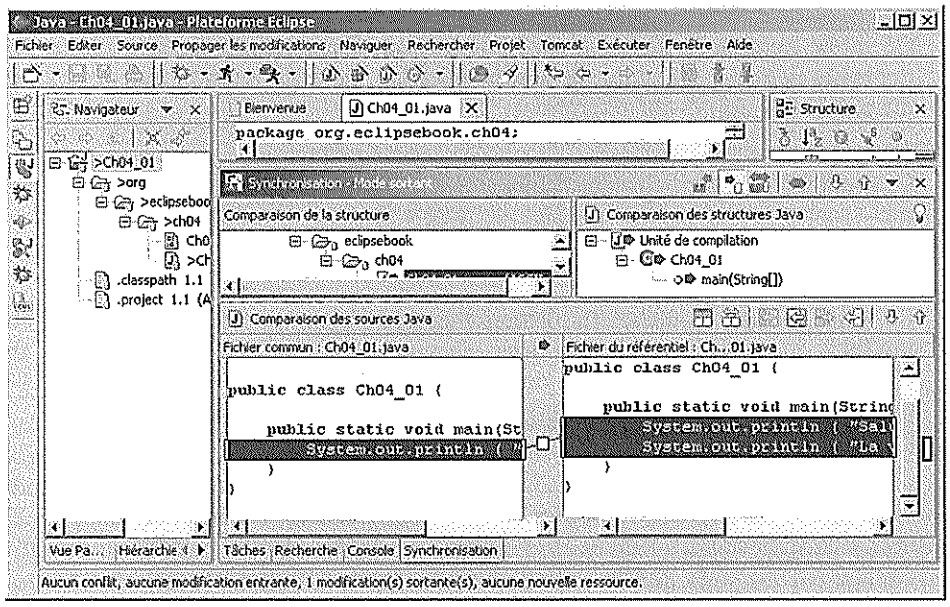


Figure 4-14. Mode sortant

> Mais en général, cela n'est pas un problème, si bien que la mise à jour et la fusion de code restent la meilleure façon de procéder avant de valider un code. Cependant, si l'on suspecte la présence de modifications très importantes, il est préférable de synchroniser le code (au lieu de faire une mise à jour) ou d'effectuer une sauvegarde du fichier juste avant de le mettre à jour.

Création d'un correctif (patch)

Les collaborateurs d'un programmeur n'ont pas tous accès au référentiel CVS. Pour leur permettre d'obtenir une version modifiée d'un code sans changement du numéro de version, il est possible de créer un *correctif* (patch). Imaginons par exemple avoir ajouté une ligne de code dans la version locale sans avoir modifié la version du référentiel :

```
public class Ch04_01 {

        public static void main(String[] args) {
                System.out.println("Salut la compagnie.");
                System.out.println("La vie est belle ?");
                System.out.println("Bien sûr.");
        }
}
```

Il est possible de créer un correctif qui va convertir le code standard de cette version en la nouvelle forme du code. Lorqu'un correctif est créé, Eclipse compare la version locale du code avec celle du référentiel : un fichier correctif contenant les différences entre ces deux versions est alors créé.

Créer un correctif permet ainsi au programmeur de transmettre plus rapidement les modifications de son code aux utilisateurs de celui-ci, sans passer par le référentiel CVS. Pour ce faire, il faut sauvegarder la version locale modifiée, cliquer le bouton droit de la souris dessus, et sélectionner Equipe → Créer un correctif, ce qui ouvre la boîte de dialogue de la figure 4-15.

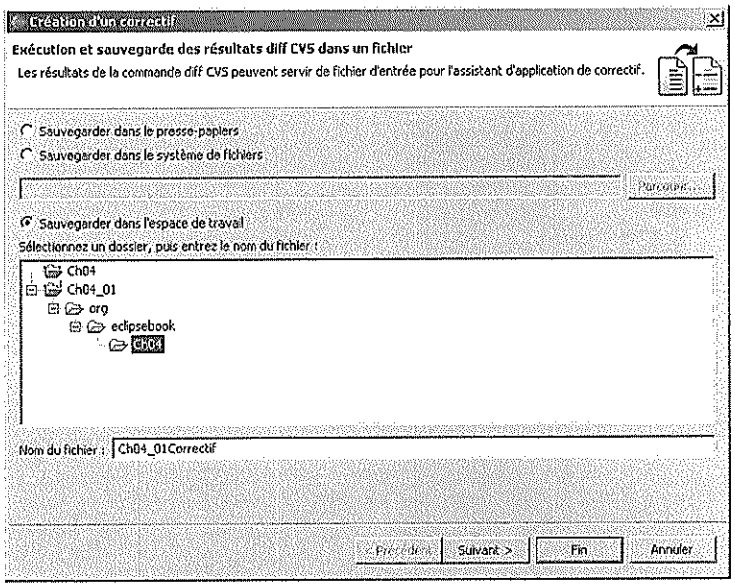


Figure 4-15. Création d'un correctif

Le fichier correctif peut être sauvegardé sous n'importe quel nom ; dans l'exemple présent, le fichier patch s'appelle *Ch04_01Correctif*. Il suffit alors de cliquer le bouton Fin pour déclencher la création de ce fichier – comme on peut le voir, il s'agit d'un simple fichier texte, qui peut être envoyé par messagerie électronique (noter que chaque début de ligne marqué par un seul signe + signifie qu'Eclipse ajoutera cette ligne au code corrigé):

```
Index: Ch04_01.java
===================================================================
RCS file: c:/referentiel/Ch04_01/org/eclipsebook/ch04/Ch04_01.java,v
retrieving revision 1.4
diff -u -r1.4 Ch04_01.java
--- Ch04_01.java        01 Jul 2004 21:13:46 -0000        1.4
+++ Ch04_01.java        01 Jul 2004 17:34:54 -0000
@@ -17,5 +17,6 @@
         public static void main(String[] args) {
                 System.out.println("Salut la compagnie.");
                 System.out.println("La vie est belle ?");
+                System.out.println("Bien sûr.");
         }
  }
```

Pour appliquer un nouveau correctif à un code non encore corrigé, il suffit de cliquer le bouton droit de la souris sur le fichier à mettre à jour sous Eclipse, et de sélec-

tionner l'option Equipe → Appliquer un correctif : la boîte de dialogue de la figure 4-16 apparaît alors.

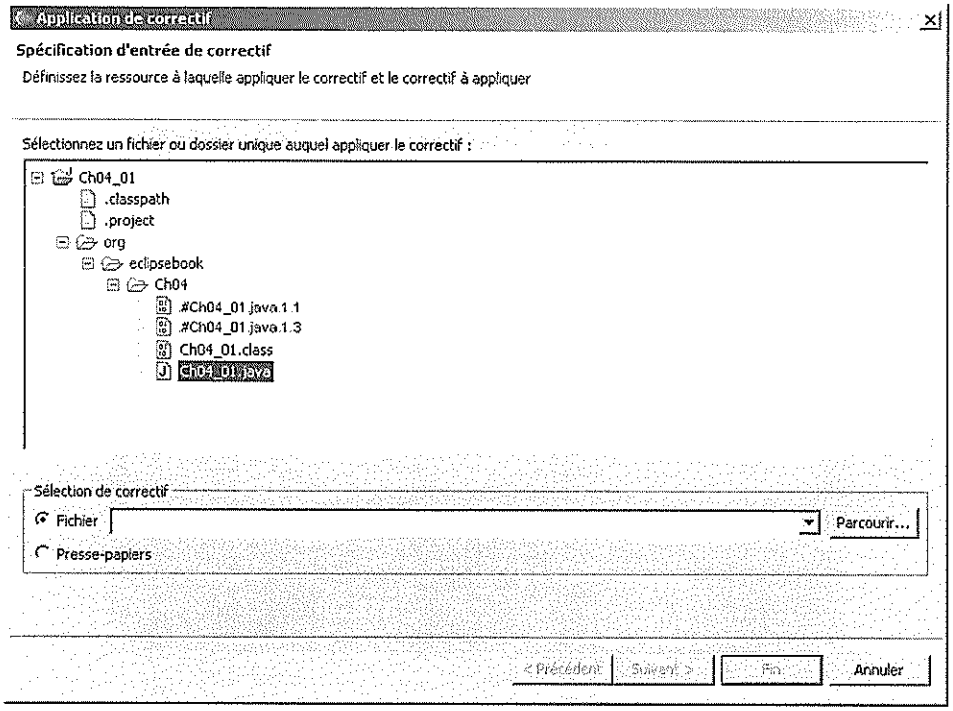


Figure 4-16. Application d'un correctif

On recherche le fichier correctif au moyen du bouton Parcourir et on clique Suivant, ce qui provoque l'ouverture de la boîte de dialogue de la figure 4-17. On peut y voir les modifications que le correctif va effectuer dans la version locale du fichier. Comme on peut le voir, Eclipse propose l'insertion de la ligne `System.out.println("Bien sur.");` à la version locale du code. On clique le bouton Fin pour appliquer le correctif.

L'application du correctif débouche sur l'insertion de la nouvelle ligne dans la version locale du fichier (voir figure 4-18). Le numéro de version du fichier n'est pas changé, alors que le code l'a été. Ainsi, Eclipse automatise intégralement le mécanisme de mise en place d'un correctif.

Balisage de version

Lorsqu'une version charnière d'un code est réalisée, il est possible de la baliser au moyen d'une étiquette de version qui permet à CVS de stocker cette version et de la rendre accessible ultérieurement. Pour baliser un projet au moyen d'une étiquette de version, il suffit de cliquer le bouton droit de la souris sur le projet et de sélectionner Equipe → Baliser en tant que version, ce qui ouvre la boîte de dialogue de la figure 4-19.

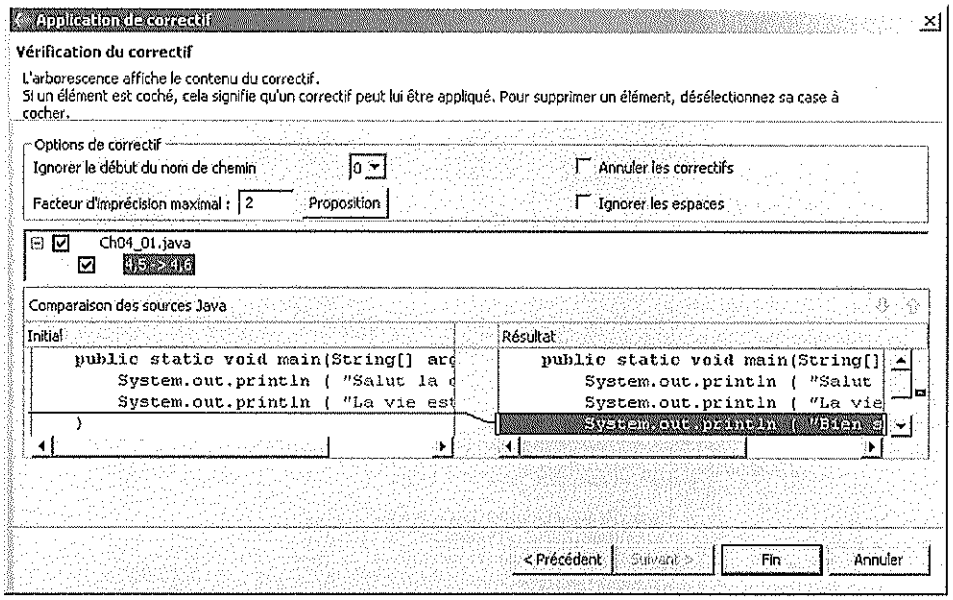


Figure 4-17. Configuration d'un patch

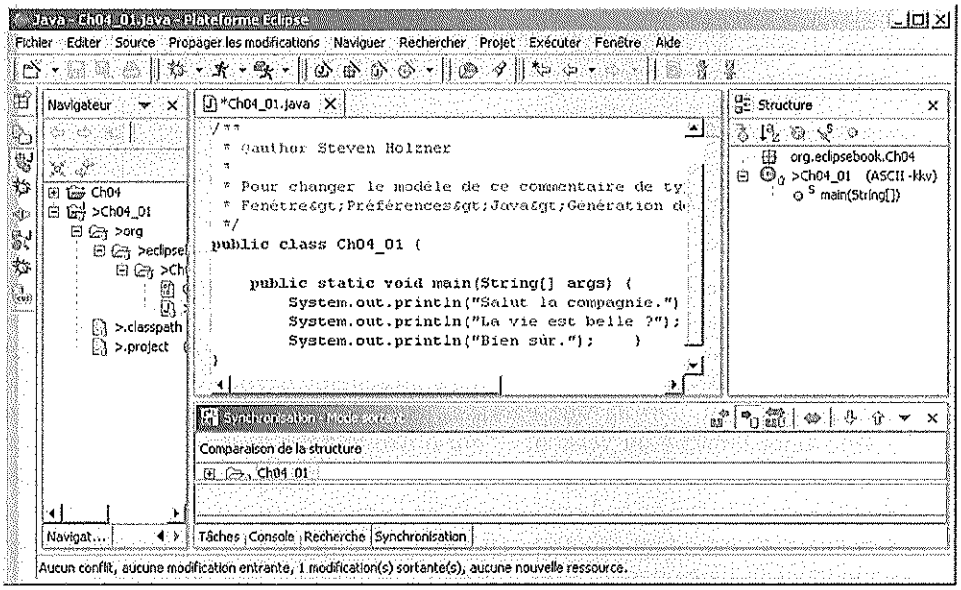


Figure 4-18. Application d'un correctif

Dans le cas présent, nous allons marquer la version courante du projet au moyen de l'étiquette `PremièreSauvegardeBalisée` : on doit donc taper taper celle-ci et cliquer OK. Les étiquettes de version doivent débuter par une lettre et ne peuvent inclure les caractères espace et suivants : '$,.:;@|'. Après avoir étiqueté la version courante, on peut la retrouver dans le nœud Versions de la vue Référentiels CVS (voir figure 4-20).

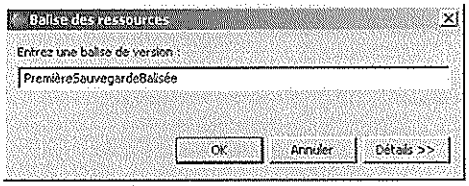


Figure 4-19. Balisage d'une version

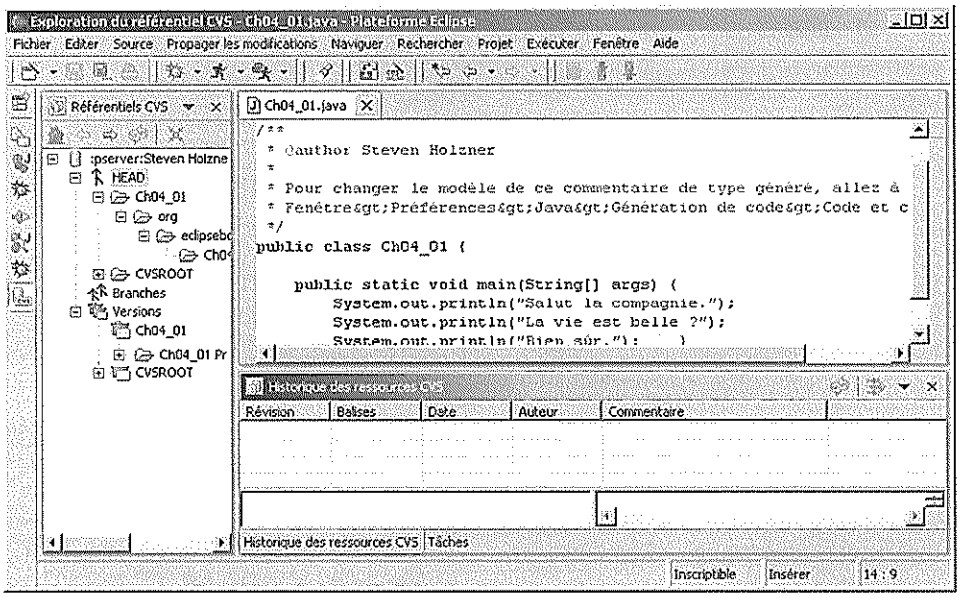


Figure 4-20. Examen d'une version étiquetée

Il est possible de réserver une version étiquetée d'un module en cliquant dessus avec le bouton droit de la souris, dans la vue Référentiels CVS, et en sélectionnant une des options du menu contextuel telle que Réserver en tant que projet, comme pour tout autre module CVS. De façon alternative, il est possible de cliquer le bouton droit de la souris sur un projet, dans la Vue Packages, et de sélectionner l'option Remplacer par → Une autre branche ou version, ce qui ouvre la boîte de dialogue de la figure 4-21. On sélectionne la version amenée à remplacer la version courante et on clique sur OK.

Le balisage d'un projet au moyen d'une étiquette de version est très utile car il permet d'identifier plus facilement une version décrite par un commentaire court.

Création de branches

En marge du stockage des versions, il est possible de créer de nouvelles branches, qui constituent des alternatives au fil principal de développement d'un code. Par exemple, on peut vouloir créer une version d'un logiciel supportant l'Internet pour pouvoir le tester, ce qui nécessite l'ajout d'une nouvelle branche à l'arborescence du fil principal de développement.

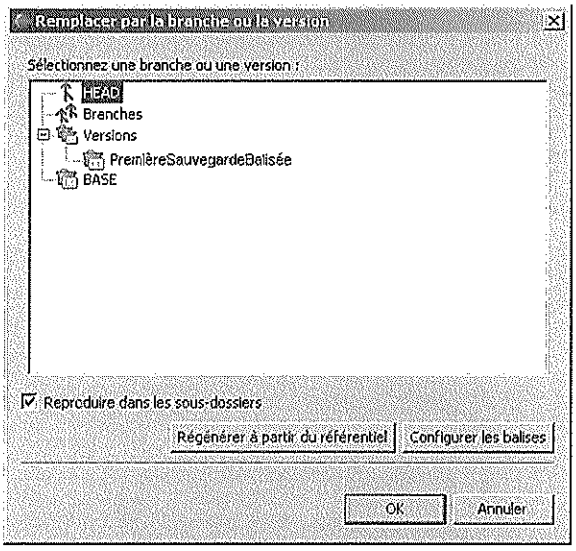


Figure 4-21. Remplacement de la version courante par une autre version

Pour créer une branche, il faut cliquer le bouton droit de la souris sur le projet concerné et sélectionner Equipe → Branche, ce qui provoque l'ouverture de la boîte de dialogue Création d'une nouvelle branche CVS, visible dans la figure 4-22. Il faut alors donner un nom à cette nouvelle branche (ici, `BrancheVersionInternet`) ; on notera que les noms des branches sont affublées des mêmes restrictions dactylographiques que les étiquettes de version. On peut aussi créer un nouveau nom de version qui servira de point de départ à la branche, ce qui donne à Eclipse un point de départ pour fusionner les branches avec le fil principal de développement.

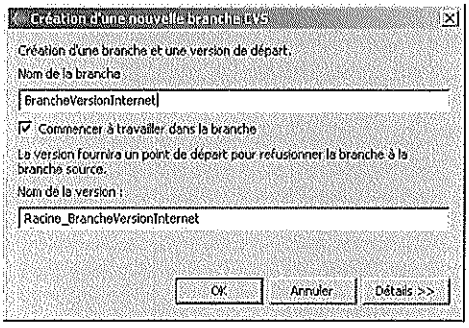


Figure 4-22. Création d'une nouvelle branche

Si la case « Commencer à travailler dans la branche » reste cochée, on commence immédiatement à travailler avec le code de cette branche. On peut aussi réserver une branche à partir de la vue Référentiels CVS, comme on peut le voir dans la figure 4-23.

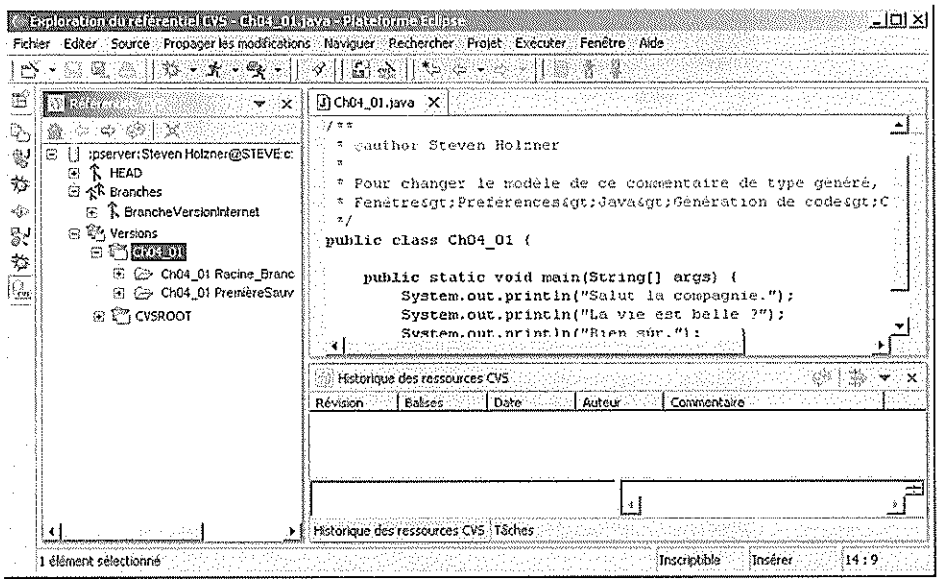


Figure 4-23. Exploration de branches

Pour fusionner une branche avec le fil principal de développement, il suffit de cliquer le bouton droit de la souris sur celle-ci, dans la Vue Packages, et de sélectionner Equipe → Fusionner, ce qui provoque l'ouverture de la boîte de dialogue Fusionner. On peut alors sélectionner le point de départ de la fusion (il s'agit du nom de version donné lors de la création de la branche) et l'on clique sur Fin.

Ainsi s'achève la présentation du travail en équipe sous CVS. Une part importante du travail d'un programmeur consiste à partager son code avec les autres membres d'une équipe. Or Eclipse est particulièrement adapté à cette problématique.

CHAPITRE 5

Compilation d'un projet Eclipse au moyen de Ant

Eclipse est particulièrement adapté à la compilation d'une application, mais il lui manque cependant quelques fonctionnalités pour développer des projets sophistiqués. Que faire, par exemple, si l'on veut non seulement compiler simultanément divers fichiers, mais aussi copier des fichiers dans d'autres répertoires de travail, créer des fichiers JAR et Javadoc, créer de nouveaux dossiers, supprimer les précédentes compilations et créer les paquetages de diffusion, le tout en une seule et unique opération ?

Tout ceci est possible avec Ant, un outil Java développé par la fondation Apache (*http://ant.apache.org/*). Ant se télécharge puis s'exécute à partir de la ligne de commande. Il permet d'automatiser toutes les tâches liées à la construction d'une application (compilation de code, création de fichiers JAR, création ou déplacement de classes, création ou destruction de dossiers, etc.).

Ant est directement installé avec Eclipse : il est donc immédiatement opérationnel. Dans ce chapitre, nous verrons que Ant est le meilleur des outils pour le développement en Java. Pour les projets les plus élaborés, Ant peut automatiser plusieurs dizaines de tâches qu'il aurait sinon fallu réaliser manuellement. Après avoir spécifié tout ce que doit réaliser Ant, la seule chose qui nous reste à faire, pour obtenir la compilation complète, se réduit à un simple clic qui évite une interminable suite d'étapes, et donc tous les risques d'oublis qui mettent en échec le processus de développement.

Le fait que Ant soit intégré à Eclipse le rend facile à utiliser et surtout nettement plus simple qu'avec la ligne de commande. Un petit exemple va le démontrer.

Travailler avec Ant

Afin de découvrir l'utilisation de Ant dans Eclipse, nous allons créer un nouveau projet, `Ch05_01`, et ajouter une nouvelle classe `Ch05_01`. Dans la méthode de classe `main`, on affiche le message « Pas de souci » comme le montre l'exemple 5-1.

Exemple 5-1. Un exemple de projet

```
package org.eclipsebook.ch05;

public class Ch05_01 {
```

Exemple 5-1. Un exemple de projet (suite)

```
        public static void main(String[] args) {
                System.out.println("Pas de souci");
        }
}
```

Pour travailler avec Ant, il faut disposer d'un fichier de compilation dont le nom par défaut est *build.xml*. Pour créer et ajouter ce fichier au projet, il suffit de cliquer le bouton droit de la souris sur le projet et sélectionner Nouveau → Fichier dans le menu contextuel. Dans la case Nom du fichier, on saisit ensuite le nom du nouveau fichier, *build.xml*, puis on clique sur Fin. On utilise XML dans ce fichier pour spécifier à Ant comment compiler le projet. Eclipse se connecte automatiquement à Ant, lequel nécessite ce fichier de compilation en XML pour savoir ce qu'il doit faire ; c'est pourquoi il faut avoir la maîtrise de la syntaxe de ce type de fichier.

Éclipse reconnaît que *build.xml* est un fichier Ant et l'indique avec l'icône de Ant, comme le montre la figure 5-1 à gauche.

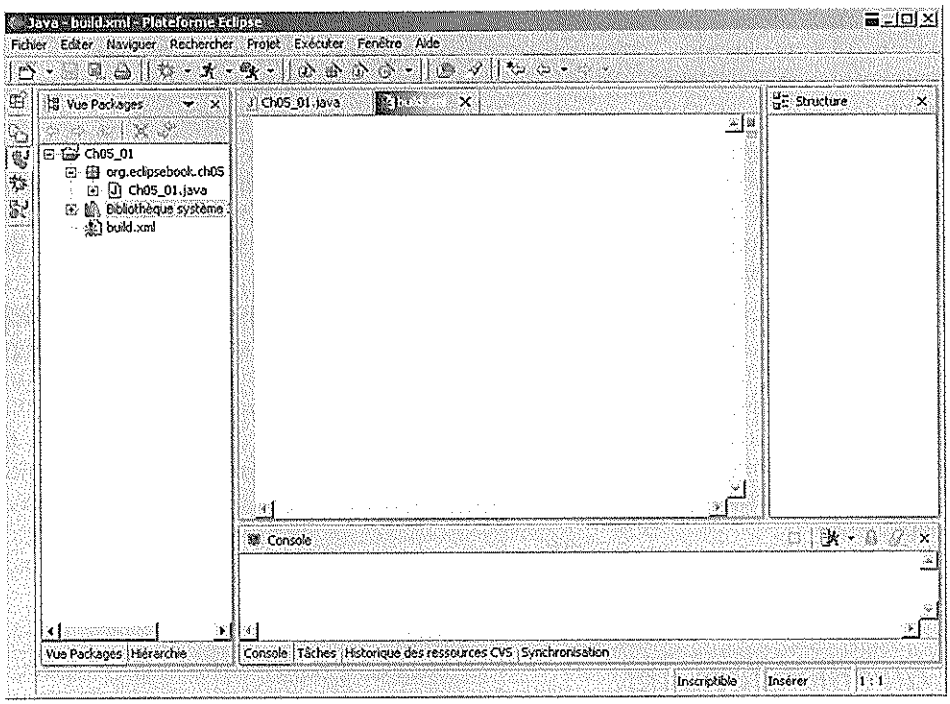


Figure 5-1. Créer un fichier build.xml

On saisit ce simple texte XML dans *build.xml*, grâce auquel on obtiendra le message « Ant au travail! », dans la console :

```
<?xml version = "1.0" encoding="UTF-8" ?>
<project name = "Ch05_01" default = "Compilation principale">
    <target name = "Compilation principale">
```

```
        <echo message = "Ant au travail!" />
    </target>
</project>
```

Ce XML crée dans la *cible par défaut* une *cible*, nommée « Compilation principale ». Une cible Ant définit un ensemble de tâches à accomplir, semblable à une méthode en Java. La cible par défaut est celle qui est exécutée quand aucune cible particulière n'est précisée à Ant. Dans le cas présent, la seule chose demandée à la cible par défaut est l'affichage d'un message à la console ; rien n'est compilé.

On peut observer le texte XML dans l'éditeur de Ant (voir figure 5-2). Cet éditeur utilise la mise en valeur syntaxique, exactement comme le fait l'éditeur JDT et l'on peut voir la structure de ce document *build.xml* dans la vue Structure à droite.

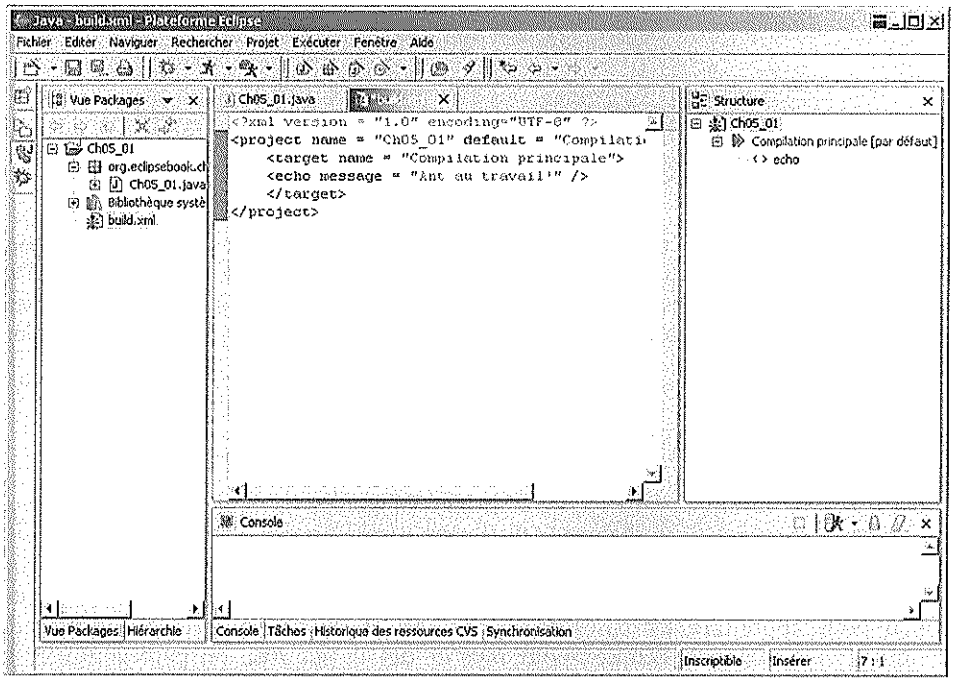


Figure 5-2. Saisie du XML dans build.xml

On enregistre *build.xml*, puis avec un clic droit sur le fichier lui-même, on sélectionne l'item Exécuter Ant. Cela ouvre la boîte de dialogue `Ch05_01` *build.xml* visible dans la figure 5-3. La cible par défaut, `Compilation Principale`, est déjà sélectionnée.

Dans cette boîte de dialogue, un clic sur le bouton Exécuter lance Ant. Dans la vue Console cela produit le résultat suivant (noter l'affichage du message prévu) :

```
Buildfile: D:\eclipse211\eclipse\workspace\Ch05_01\build.xml

Main Build:
       [echo] Ant au travail!
BUILD SUCCESSFUL
Total time: 430 milliseconds
```

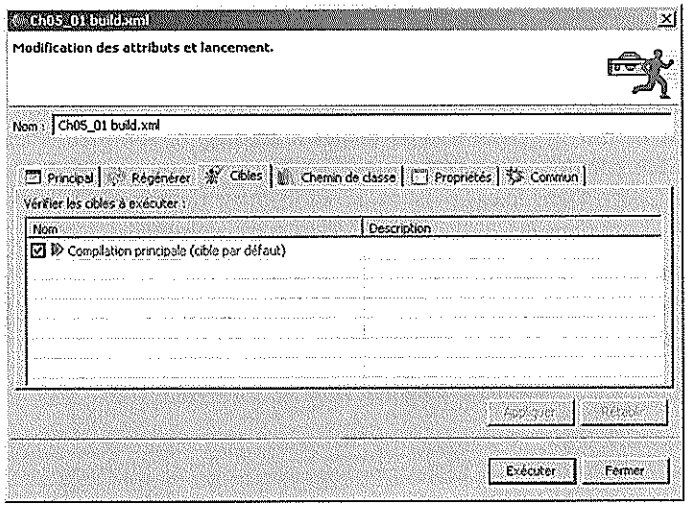


Figure 5-3. Exécution de Ant

C'est exactement ce qui était demandé dans ce cas particulier. Ce petit exemple montre comment Ant interagit avec Eclipse, mais il ne fait qu'afficher « Ant au travail! ». Il ne compile rien, ne crée aucun fichier JAR, mais il montre comment démarrer avec Ant dans Eclipse. L'exemple qui suit permet de découvrir d'autres possibilités bien plus évoluées.

Production de fichiers JAR

Voici un autre exemple dans lequel le projet Eclipse est compilé et le fichier *.class* résultant est stocké dans un fichier JAR. Il n'est pas nécessaire d'être un professionnel de Ant pour comprendre ce qui suit, notre but étant seulement de découvrir Ant selon le point de vue d'Eclipse, et non pas les détails de Ant *per se*. Cet exemple est conçu pour apprendre les bases de la création et de l'exécution d'un fichier de compilation Ant depuis Eclipse ; pour plus de détails sur Ant, on peut consulter le manuel à l'adresse *http://ant.apache.org/manual/index.html*.

Ici, le but est de créer un projet Java dans Eclipse, d'utiliser Ant pour le compiler et stocker le fichier *.class* résultant dans un fichier JAR. Pour ce faire, on crée un nouveau projet Java, `Ch05_02`. Afin de simuler une situation de projet réel, le code source de l'exemple est stocké dans un dossier nommé *src* et les fichiers de sortie le sont dans le dossier nommé *bin*. C'est dans le troisième panneau du dialogue Nouveau Projet Java que l'on peut créer ces répertoires. On clique sur l'onglet Source puis sur le bouton Ajouter un Dossier, ce qui ouvre la boîte de dialogue Sélection du dossier source, dans laquelle on clique sur le bouton Créer un dossier afin d'accéder au dialogue Nouveau dossier. On saisit alors le nom `src` dans le champ éditable puis l'on clique deux fois sur OK. Eclipse demande de confirmer le remplacement du dossier projet comme dossier source et l'utilisation du dossier nommé *Ch05_02/bin* comme dossier de sortie. Pour créer enfin ce nouveau projet, avec ses deux dossiers *src* et *bin*, on confirme par un clic sur Oui puis sur Fin.

On ajoute ensuite à ce projet une nouvelle classe, Ch05_02, dans un paquetage nommé org.eclipsebook.ch05. On accepte la mise en place de la méthode main pour pouvoir afficher le message « Ce code a été créé avec Ant. », comme le montre l'exemple 5-2.

Exemple 5-2. Exemple de projet

```
package org.eclipsebook.ch05;

public class Ch05_02 {

    public static void main(String[] args) {
        System.out.println("Ce code a été créé avec Ant.");
    }
}
```

Finalement, on incorpore *build.xml* au projet grâce au clic droit sur le projet dans la Vue Packages et à la sélection de Nouveau → Fichier. Dans la case Nom de fichier, on inscrit *build.xml* puis l'on clique sur Fin, ce qui crée le fichier et ouvre l'éditeur Ant. L'édition de *build.xml* commence avec les déclarations XML standard et la balise <project> indique que la tâche par défaut se nomme Compilation principale :

```
<?xml version="1.0" encoding = "UTF-8"?>
<project name="Ch05_02" default="Compilation principale" basedir=".">
        .
        .
        .
</project>
```

Il faut ensuite définir les propriétés associées aux dossiers spécifiés – src, bin, le dossier pour les fichiers JAR, jardir (y compris la création d'un dossier *lib* à l'intérieur du dossier *bin* pour y stocker les fichiers JAR), et le fichier JAR lui-même, jarfile (ce fichier sera nommé *Ch05_02.jar*). En fixant, dans ce fichier *build.xml*, toutes ces propriétés, il sera possible d'accéder à ces dossiers par leur nom. Il faut aussi renseigner la propriété build.compiler pour y attacher l'adaptateur du compilateur JDT, org.eclipse.jdt.core.JDTCompilerAdapter, que Ant utilisera :

```
<?xml version="1.0" encoding = "UTF-8"?>
<project name="Ch05_01" default="Compilation principale" basedir=".">

    <property name="bin" location="bin"/>
    <property name="src" location="src"/>
    <property name="jardir" location="${bin}/lib"/>
    <property name="jarfile" location="${jardir}/Ch05_02.jar"/>
    <property name="build.compiler"
        value="org.eclipse.jdt.core.JDTCompilerAdapter"/>
        .
        .
        .
</project>
```

À présent, il reste à définir la tâche principale, Compilation principale. Dans le fichier Ant, on distingue trois parties – une partie initialisation, une partie compilation et une partie pour la création du fichier JAR – chacune ayant sa propre tâche : Initialisation, Compilation et Jar. Afin de s'assurer que toutes ces tâches

sont parfaitement exécutées, on précise que la tâche principale par défaut dépend de chacune d'elles en utilisant l'attribut depends. Seule la tâche par défaut inscrit un message sur la console indiquant que Ant est au travail – chacune des tâches étant soigneusement exécutée par Ant :

```
<?xml version="1.0" encoding = "UTF-8"?>
<project name="Ch05_01" default="Compilation principale" basedir=".">

    <property name="bin" location="bin"/>
    <property name="src" location="src"/>
    <property name="jardir" location="${bin}/lib"/>
    <property name="jarfile" location="${jardir}/Ch05_02.jar"/>
     <property name="build.compiler"
        value="org.eclipse.jdt.core.JDTCompilerAdapter"/>

    <target name="Compilation principale"
            depends="Initialisation, Compilation, Jar">
        <echo message="Ant au travail!"/>
    </target>
        .
        .
        .
</project>
```

La tâche `Initialisation` détruit tous les dossiers de sortie `${bin}` et `${jardir}`, puis les recrée :

```
<target name="Initialisation">
    <delete dir="${bin}"/>
    <delete dir="${jardir}"/>
    <mkdir dir="${bin}"/>
    <mkdir dir="${jardir}"/>
</target>
```

La tâche de `Compilation` compile le fichier source présent dans le dossier `${src}` (précisément *Ch05_02.java*) et place le fichier résultant *.class* dans le dossier `${bin}`:

```
<target name="Compilation" depends="Initialisation">
    <javac srcdir="${src}"
        destdir="${bin}">
    </javac>
</target>
```

Finalement, la tâche `Jar` compresse *Ch05_02.class* dans le fichier JAR et le stocke en tant que `${jarfile}` – il convient de noter que cette tâche dépend des tâches d'`Initialisation` et de `Compilation` :

```
<target name="Jar" depends="Initialisation, Compilation">
    <jar destfile="${jarfile}" basedir="${bin}"/>
</target>
```

Cette dernière étape complète *build.xml*. On peut visualiser la totalité du fichier dans l'exemple 5-3.

Exemple 5-3. Un exemple de fichier Ant de construction

```
<?xml version="1.0" encoding = "UTF-8"?>
<project name="Ch05_02" default="Compilation principale" basedir=".">
```

Exemple 5-3. Un exemple de fichier Ant de construction (suite)

```
    <property name="bin" location="bin"/>
    <property name="src" location="src"/>
    <property name="jardir" location="${bin}/lib"/>
    <property name="jarfile" location="${jardir}/ch05_01.jar"/>
     <property name="build.compiler"
        value="org.eclipse.jdt.core.JDTCompilerAdapter"/>

    <target name="Compilation principale" depends="Initialisation, Compilation,
                          Jar">
        <echo message="Ant au travail!"/>
    </target>

    <target name="Initialisation">
        <delete dir="${bin}"/>
        <delete dir="${jardir}"/>
        <mkdir dir="${bin}"/>
        <mkdir dir="${jardir}"/>
    </target>

    <target name="Compilation" depends="Initialisation">
        <javac srcdir="${src}"
            destdir="${bin}">
        </javac>
    </target>

    <target name="Jar" depends="Initialisation, Compilation">
        <jar destfile="${jarfile}" basedir="${bin}"/>
    </target>

</project>
```

Dans certaines circonstances, Eclipse peut générer le script Ant. Si le projet possède déjà un fichier manifeste, comme les projets de plug-in présentés au chapitres 11 et 12 (le fichier de plug-in étant nommé *plugin.xml*), il suffit d'ouvrir le menu contextuel avec un clic droit sur le fichier manifeste et de sélectionner Créer le fichier de compilation Ant.

Quand on saisit le code XML dans *build.xml*, on peut voir ses propriétés et ses tâches dans la vue Structure, à l'image de la figure 5-4.

Pour Compiler le projet, sur *build.xml*, dans la Vue Packages, il faut sélectionner avec un clic droit l'option Exécuter Ant. Cela ouvre la boîte de dialogue `Ch05_02` *build.xml* visible sur la figure 5-5. On y trouve les différentes cibles disponibles, compilables séparément. La cible par défaut, `Compilation Principale`, étant déjà sélectionnée, il suffit de cliquer sur le bouton Exécuter pour compiler le projet.

On peut observer le résultat de la compilation sur la figure 5-6 – compilation réussie, comme le montre la vue de la Console.

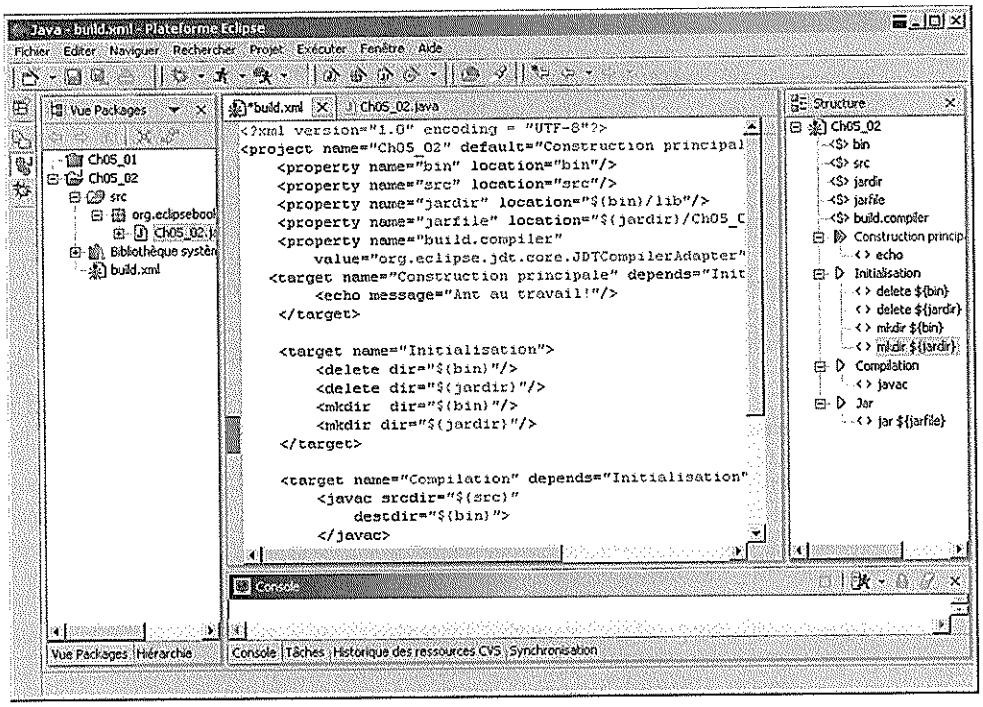


Figure 5-4. Le fichier build.xml

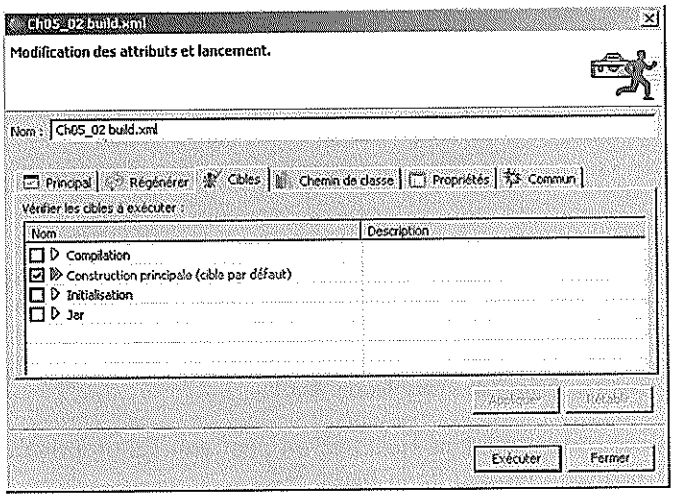


Figure 5-5. Sélection de la cible à exécuter

Voici la totalité du texte affiché dans la Console – on peut y voir le résultat de l'exécution de chaque tâche :

```
Buildfile: C:\eclipse\workspace\Ch05_02\build.xml
```

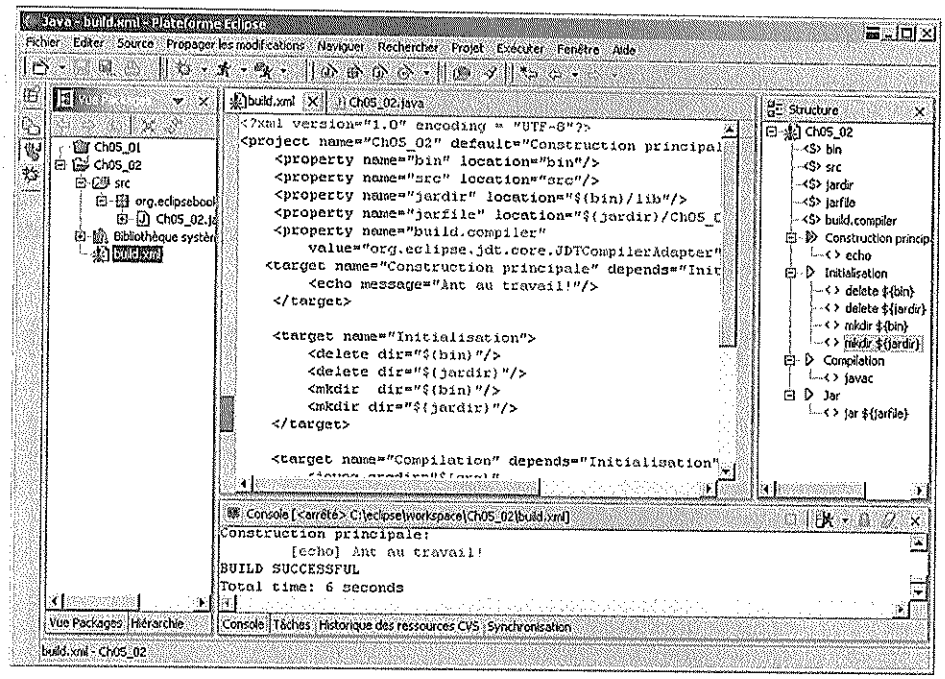


Figure 5-6. Une compilation réussie

```
Initialisation:
      [delete] Deleting directory C:\eclipse\workspace\Ch05_02\bin
       [mkdir] Created dir: C:\eclipse\workspace\Ch05_02\bin
       [mkdir] Created dir: C:\eclipse\workspace\Ch05_02\bin\lib

Compilation:
      [javac] Compiling 1 source file to C:\eclipse\workspace\Ch05_02\bin
      [javac] 8 lignes compilées en 3345 ms (2.3 lignes/s)
      [javac] 1 fichier .class généré

Jar:
        [jar] Building jar: C:\eclipse\workspace\Ch05_02\bin\lib\Ch05_02.jar

Construction principale:
       [echo] Ant au travail!
BUILD SUCCESSFUL
Total time: 4 seconds
```

Le projet a été compilé et *Ch05_02.jar* a été créé dans le dossier *bin/lib*. Comme on peut le voir, la copie de fichiers, la création de dossiers et la compression de fichiers JAR, entre autres, font qu'une compilation Ant dépasse largement le processus normal de compilation d'Eclipse.

Configuration de Ant dans Eclipse

Eclipse permet aussi de configurer sa propre version de Ant. Ainsi, pour configurer la façon dont s'exécute Ant, on peut sélectionner Fenêtre → Préférences, puis l'option Ant comme le montre la figure 5-7. Dans Eclipse, il n'est pas obligatoire de nommer le fichier de compilation *build.xml* ; Ant doit tout de même savoir quel est le fichier de compilation (c'est un fichier XML qui doit posséder l'extension *.xml*). Dans cette boîte de dialogue, on peut aider Eclipse à gérer la situation en donnant un nom alternatif ou une liste de noms.

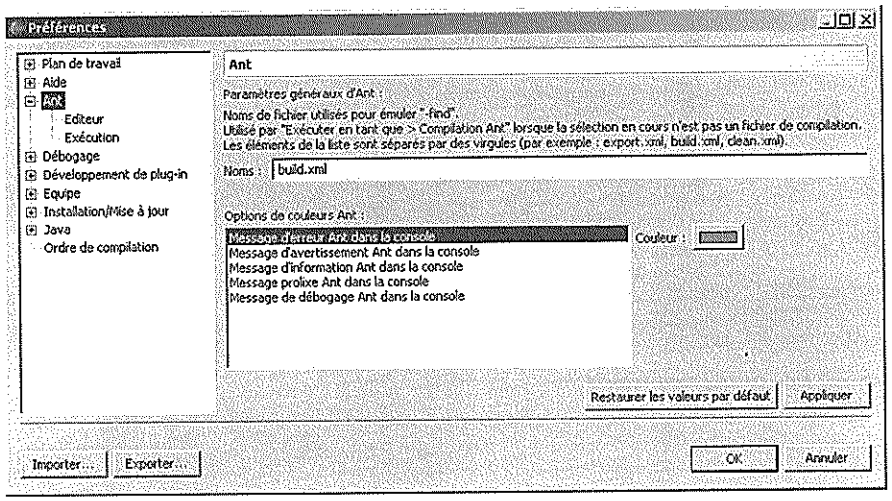


Figure 5-7. Configuration du fichier de compilation

En sélectionnant l'option Exécution, on peut aussi choisir des modes d'exécution de Ant (voir figure 5-8). On peut, par exemple, utiliser une version plus récente de Ant, il faut sélectionner l'option Exécution et modifier les lignes JAR comme le montre la figure 5-8 avec les nouvelles versions de *ant.jar* et de *optional.jar*. (Eclipse 2.1.1 s'installe avec Ant 1.5.3, mais à l'instant où ces lignes sont écrites, la dernière version de Ant est la 1.5.4 et la version 1.6 est en phase bêta. Il est possible de se procurer une autre version sur le site *http://ant.apache.org*.) Dans cette boîte de dialogue, il est aussi possible de modifier une variable Ant comme par exemple `ANT_HOME`.

Avec les onglets Tâches et Types, on peut ajouter de nouvelles tâches et d'autres types ; après quoi, ces tâches et ces types sont valides pendant la compilation des fichiers, sans avoir à utiliser les mots `taskdef` ou `typedef`. Dans cette même boîte de dialogue, si l'on choisit l'onglet Propriétés, on peut donner des valeurs à des propriétés globales de Ant. Dans cet onglet, on peut aussi ajouter une nouvelle propriété globale, en cliquant sur le bouton Ajouter, à laquelle on attribue un nom et une valeur.

On dispose aussi d'un certain contrôle sur les options de l'éditeur de Ant en sélectionnant l'option Ant → Editeur dans la boîte de dialogue Préférences. L'éditeur actuel de Ant est légèrement plus simple qu'un éditeur XML, mais on peut tout de

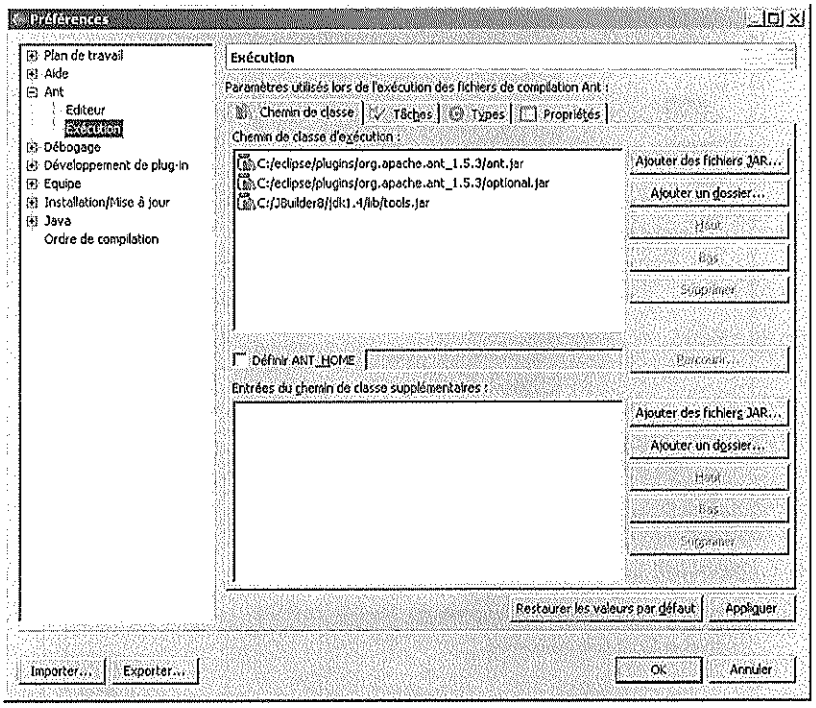


Figure 5-8. Configuration de Ant dans Eclipse

même modifier la colorisation syntaxique des éléments afficher la règle, comme le montre la figure 5-9.

L'éditeur de Ant dispose d'un assistant au codage. Par exemple, si l'on attend après la frappe d'un <, l'assistant au codage propose une liste des éléments syntaxiques possibles dans un fichier de compilation Ant. Le fait de taper une lettre fait descendre dans la liste – par exemple, frapper <p fait que l'assistant offre le choix entre <path>, <patternset> et <property>. L'assistant au codage propose aussi la liste des attributs des mots réservés de Ant : il suffit de cliquer dans la balise et de presser Ctrl+Espace. De même, le maintien sur un item de la liste ouverte par l'assistant au codage force Eclipse à afficher une explication sur ce que fait cet item. Par exemple, l'explication pour le mot property est « Sets a property by name, or a set of properties (from file or resource) in the project. »

Ayant sélectionné une cible à compiler, on peut configurer Ant juste avant l'exécution. Avec un clic droit sur *build.xml*, on doit sélectionner Exécuter Ant, et dans la boîte de dialogue ainsi ouverte, cliquer sur l'onglet Principal, comme le montre la figure 5-10. On peut alors modifier la position du fichier de compilation à utiliser, tout comme le Répertoire de base pour la compilation. On peut aussi donner des arguments à Ant dans le champ Arguments.

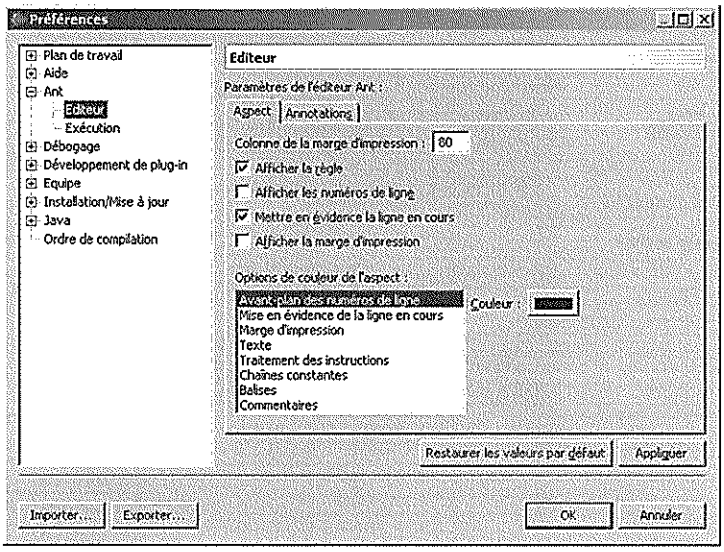


Figure 5-9. Configuration de l'éditeur de Ant

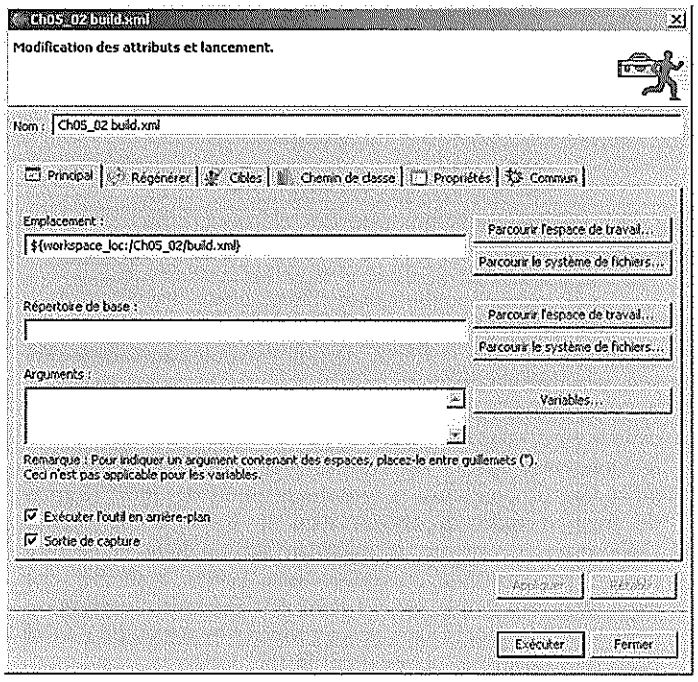


Figure 5-10. Configuration de l'exécution

Il est possible d'exécuter Ant en tant qu'outil externe : pour cela, il faut sélectionner Exécuter → Outils Externes → Outils Externes, choisir l'option Programme, cliquer sur le bouton

Nouveau et entrer le nom de la version externe de Ant. Pour remplir le champ Emplacement, on clique sur le bouton Parcourir le système de fichiers afin d'y trouver le bon fichier correspondant à l'actuel système d'exploitation (par exemple *ant.bat* dans le dossier *bin* de Ant sous Windows). Dans le champ Répertoire de travail, on saisit le répertoire du fichier de compilation et l'on clique sur Exécuter afin de lancer la compilation.

Enfin, il y a dans Eclipse une vue spécifique à Ant, que l'on ouvre au moyen de Fenêtre → Afficher la vue → Autre → Ant. Cette fenêtre visualise le fichier de compilation selon Ant ; pour ajouter un fichier de compilation à cette vue, il faut ouvrir le menu contextuel avec un clic droit et naviguer jusqu'au fichier de compilation à visualiser. Cette vue présente la structure du fichier de compilation, comme le montre la figure 5-11 ; on peut, par un simple clic droit, sélectionner et exécuter l'une des cibles.

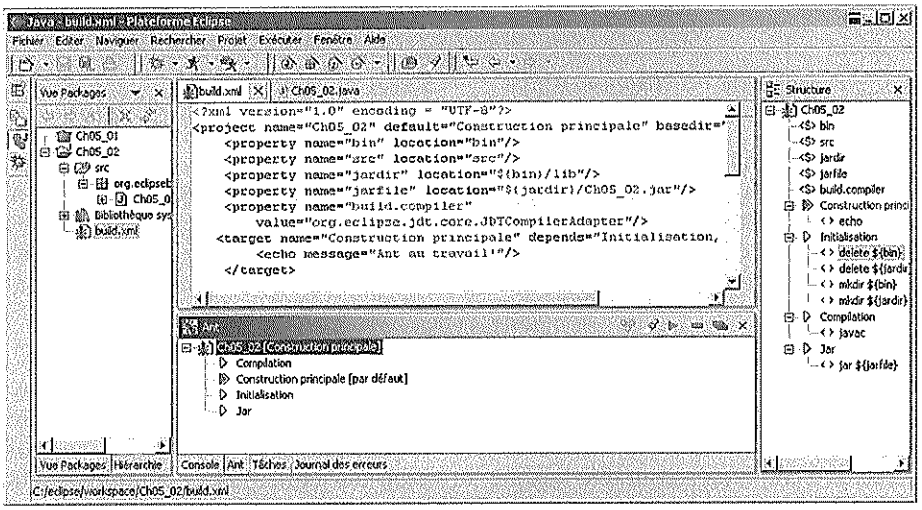


Figure 5-11. La vue de Ant

Détection des erreurs dans les fichiers de compilation

Eclipse offre quelques solutions de base pour détecter les erreurs pendant l'édition du fichier Ant. L'éditeur Ant d'Eclipse ne traite pas les erreurs de syntaxe comme le fait l'éditeur JDT pour Java – ainsi, si l'on se trompe sur l'écriture d'un attribut de `projet` en écrivant `deefault` au lieu de `default`, l'éditeur ne proteste. Par contre, s'il y a une erreur de syntaxe XML dans le fichier de compilation, comme la non fermeture d'une balise ou l'usage incorrect d'un terme (au sens où l'entend XML, c'est-à-dire si le texte du fichier de compilation n'est pas *bien écrit*), il apparaît dans l'éditeur Ant, exactement comme dans l'éditeur JDT, un soulignement avec des vaguelettes rouges et un rectangle d'explication qui indique l'erreur de syntaxe, comme le montre la figure 5-12. On identifie l'erreur grâce aux vaguelettes et à la bulle d'aide ; ainsi, sur cette figure, on constate que la balise `mkdir` n'a pas été fermée.

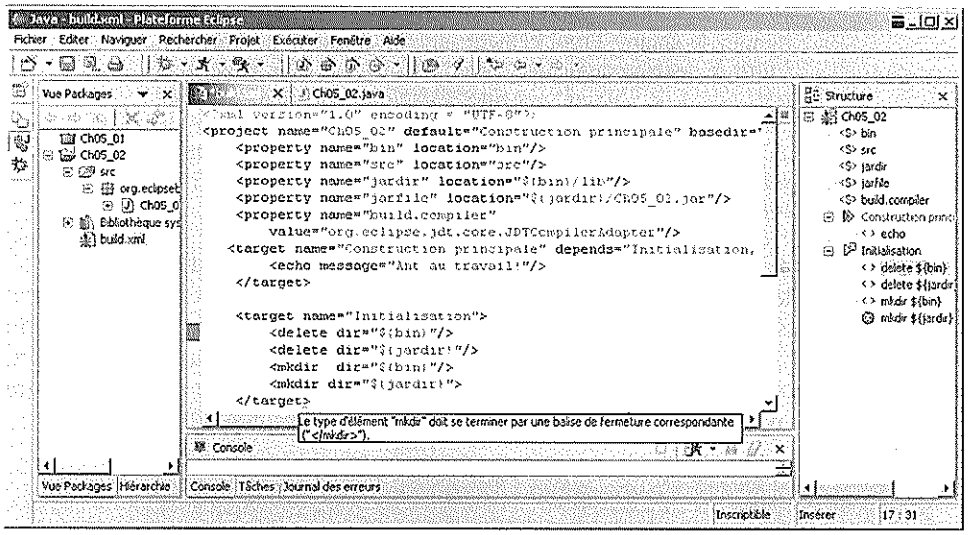


Figure 5-12. Traitement des erreurs de syntaxe

Si l'on oublie une erreur XML comme celle-ci, un message surgit instantanément à la compilation du projet pour le rappeler au programmeur. Certes, l'éditeur de Ant ne sait pas détecter les erreurs qui ne relèvent pas de XML (telle que l'attribut de `projet` noté par erreur `deefault`), mais dès qu'on lance l'exécution de Ant, il affiche automatiquement la liste des erreurs de syntaxe, comme on le voit en haut de la figure 5-13.

Malheureusement, c'est là tout ce qu'offre Eclipse pour le traitement des erreurs de Ant. Lorsque Ant s'exécute, Eclipse se contente d'afficher ses propres erreurs dans la Console (avec malgré tout des liens hyper-texte entre les erreurs et les lignes du fichier de compilation ou du fichier Java). Eclipse ne possède pas d'outil de mise au point interactif, ô combien secourable lorsque les scripts deviennent assez complexes. Étant donné la popularité de Ant, on peut espérer que les futures versions d'Eclipse intègrent un assistant pour Ant, appelé à partir de Fichier → Nouveau, qui permettra de choisir les cibles, les répertoires et les tâches à exécuter par Ant, de façon aussi aisée que la création d'un projet dans Eclipse.

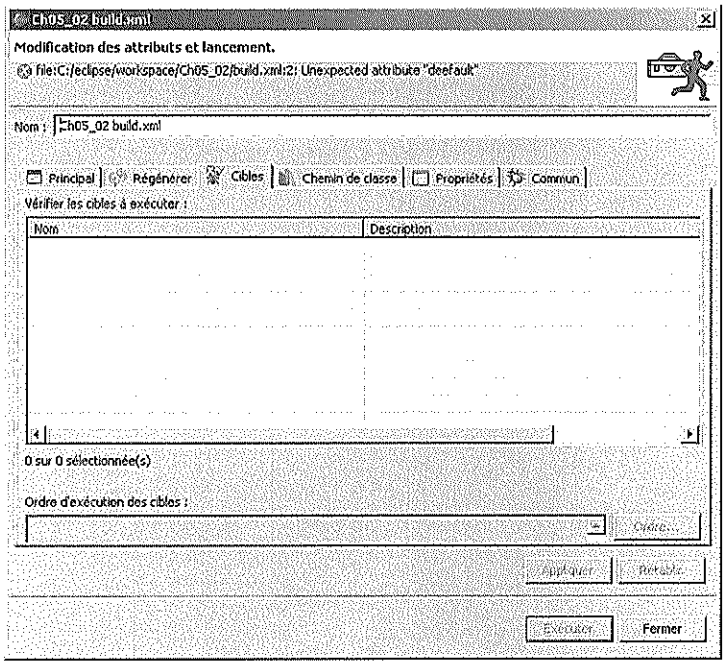


Figure 5-13. Description d'une erreur

CHAPITRE 6

Programmation GUI : des applets à Swing

Ce chapitre est totalement consacré à la création de GUI (interfaces graphiques utilisateur) en Java avec Eclipse. Jusqu'à présent, les programmes affichaient seulement du texte dans la console. Dans ce chapitre, nous aborderons la création de GUI à l'aide d'applets, du kit AWT (Abstract Windowing Toolkit) et de Swing. On verra aussi l'utilisaton d'un plug-in (module additionnel) d'Eclipse pour créer du code Swing.

Examinons tout d'abord la création des applets. Les applets Java sont apparues avec l'émergence de Java sur Internet et sont restées populaires un certain temps. Les navigateurs ont intégré Java et par suite ont supporté les applets. Depuis, les applets sont devenus moins populaires à cause de leurs limitations et sont aujourd'hui largement dépassées par des solutions « glamour » telles que Flash ou Java Web Start. Néanmoins, Eclipse dispose de moyens particuliers pour développer et tester les applets ; c'est pourquoi nous les présenterons rapidement afin de voir comment on peut s'en servir.

A partir d'Eclipse, la création d'une applet se fait exactement comme n'importe quel projet Java. Ici, on veut que l'applet affiche simplement le message « Hello depuis Eclipse! » Pour ce faire, on crée un nouveau projet Java nommé `Ch06_01`, et on y place une nouvelle classe, `Ch06_01`, dans le paquetage `org.eclipsebook.ch06`. Pour créer une applet, il faut importer `java.applet.Applet` et `java.awt.*`:

```
import java.applet.Applet;
import java.awt.*;
```

La classe `Ch06_01` hérite de la classe `Applet`:

```
public class Ch06_01 extends Applet {
        .
        .
        .
```

Dans une applet, c'est la méthode `init` qui assure l'intialisation de l'applet ; elle fixe ici un fond blanc. Les méthodes `start`, `stop` et `destroy` prennent en charge les événements liés au cycle de vie de l'applet. L'affichage d'un texte dans la fenêtre de l'applet se traduit par le passage d'un objet `Graphics` à la méthode `paint`. Voici le

code à ajouter :

```
public class Ch06_01 extends Applet {

    public void init()
    {
        setBackground(Color.white);
    }

    public void start()
    {
    }

    public void paint(Graphics g)
    {
        g.drawString("Hello depuis Eclipse!", 60, 100);
    }

    public void stop()
    {
    }

    public void destroy()
    {
    }
}
```

Voilà, c'est tout ce que nécessite l'écriture de cette applet. Afin de le tester, on sélectionne Exécuter en tant que → Applet Java, et l'applet s'exécute. Si Eclipse ne reconnaît pas l'applet, c'est qu'il faut préparer de façon spécifique la configuration pour l'exécution de ce code. Pour cela, on sélectionne le projet `Ch06_01` dans la Vue Packages et l'on choisit le menu Exécuter → Exécuter pour obtenir la boîte de dialogue visible dans la figure 6-1. Dans la boîte de dialogue Exécuter, un double clic sur Applet Java ajoute automatiquement le projet `Ch06_01` en tant qu'applet. Le clic sur le bouton Exécuter lance l'applet. Après avoir préparé la configuration, il est possible d'utiliser simplement Exécuter en tant que → Applet Java pour l'exécution de l'applet.

On visualise le résultat dans l'AppletViewer (voir figure 6-2). L'applet est exécutée conformément aux attentes.

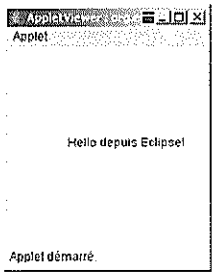


Figure 6-2. Exécution d'une applet

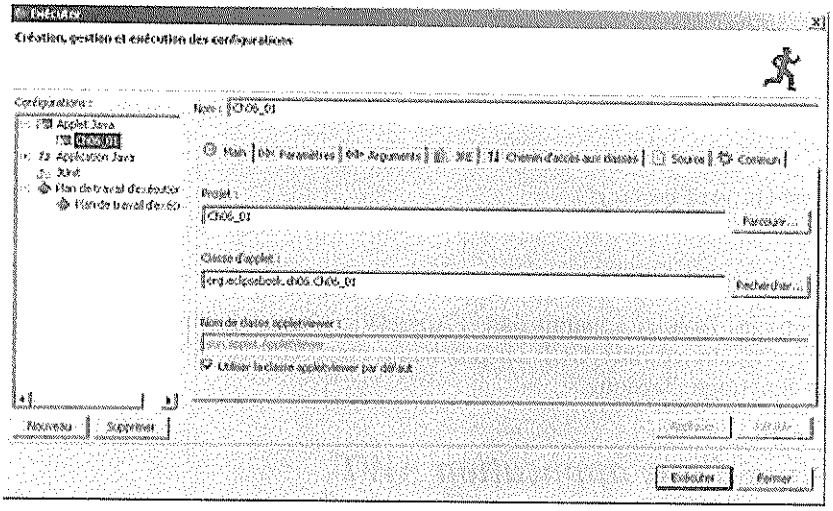


Figure 6-1. Configuration du lancement de l'applet

Pour utiliser l'applet dans une page web, on assemble le fichier *.class* généré (*Ch06_01.class*), avec la balise HTML <APPLET>, par exemple comme ceci :

```
<APPLET
    CODE = "Ch06_01.class"
    WIDTH = 300
    HEIGHT = 200
>
```

C'est tout ce qu'il faut faire ; l'applet est complète. On peut voir la totalité du code de l'applet dans l'exemple 6-1.

Exemple 6-1. Une applet simple

```
package org.eclipsebook.ch06;

import java.applet.Applet;
import java.awt.*;

public class Ch06_01 extends Applet {
    public void init()
    {
        setBackground(Color.white);
    }

    public void start()
    {
    }

    public void paint(Graphics g)
    {
        g.drawString("Hello depuis Eclipse!", 60, 100);
    }
```

Exemple 6-1. Une applet simple (suite)

```
    public void stop()
    {
    }

    public void destroy()
    {
    }
}
```

Au lancement, l'applet apparaît dans l'AppletViewer par défaut. C'est un bon début mais il y a mieux comme GUI. La prochaine étape consiste à créer une application avec un fenêtrage classique.

Création d'une application AWT

L'Abstract Windowing Toolkit, ou AWT, a été la toute première boîte à outils Java pour les GUI. Elle continue à être maintenue et utilisée ; c'est la raison pour laquelle nous présenterons ici un exemple qui s'exécute dans sa propre fenêtre. Le paquetage originel de AWT a été écrit en six semaines seulement, et les contrôles, conçus pour les applets, l'ont été après les contrôles HTML. Depuis, c'est Swing qui est devenu l'interface Java standard – Swing étant lui-même construit à partir d'AWT.

Cet exemple AWT ouvre sa fenêtre et y affiche le même message que l'applet précédente. On crée donc un nouveau projet `Ch06_02`, avec une classe `Ch06_02`, munie d'une méthode `main`, et stockée dans le paquetage `org.eclipsebook.ch06`. Il faut penser à ajouter les clauses `import java.awt.*` afin d'accéder aux services AWT et `java.awt.event.*` pour le traitement de l'événement fermeture-fenêtre :

```
import java.awt.*;
import java.awt.event.*;
```

La fenêtre, que doit afficher cet exemple, est une classe nommée `AppFrame`, basée sur la classe Frame de AWT ; on écrit alors le code suivant dans le fichier *Ch06_02.java*:

```
class AppFrame extends Frame
{
        .
        .
        .
}
```

Tout comme l'applet, on surcharge la méthode `paint` et on transmet un objet `Graphics` pour le texte que l'on veut afficher dans la nouvelle fenêtre:

```
class AppFrame extends Frame
{
    public void paint(Graphics g)
    {
        g.drawString("Hello depuis Eclipse!", 60, 100);
    }
}
```

À présent, dans la méthode main de la classe Ch06_02, on instancie la nouvelle fenêtre de type AppFrame et on en modifie la taille :

```
public class Ch06_02 {

    public static void main(String [] args)
    {
       AppFrame f = new AppFrame();

       f.setSize(200, 200);
        .
        .
        .
    }
}
```

Le reste correspond au code classique de capture de l'événement de fermeture de la fenêtre, obtenu par un clic sur le bouton X en haut à droite de la fenêtre et la méthode show pour effectivement voir la fenêtre:

```
public class Ch06_02 {

    public static void main(String [] args)
    {
       AppFrame f = new AppFrame();

       f.setSize(200, 200);

       f.addWindowListener(new WindowAdapter() { public void
           windowClosing(WindowEvent e) {System.exit(0);}});

       f.show();
    }
}
```

Pour exécuter cette application, on doit sélectionner Exécuter en tant que → Application Java. Cette application ouvre une fenêtre AWT contenant le message que l'on voit dans la figure 6-3.

Figure 6-3. Application utilisant une fenêtre AWT

C'est terminé – le code de l'application apparaît dans l'exemple 6-2.

Exemple 6-2. Une application AWT de base

```
package org.eclipsebook.ch06;
```

Exemple 6-2. Une application AWT de base (suite)

```
import java.awt.*;
import java.awt.event.*;

public class Ch06_02 {

    public static void main(String [] args)
    {
       AppFrame f = new AppFrame();

       f.setSize(200, 200);

       f.addWindowListener(new WindowAdapter() { public void
           windowClosing(WindowEvent e) {System.exit(0);}});

       f.show();
    }
}

class AppFrame extends Frame
{
    public void paint(Graphics g)
    {
        g.drawString("Hello depuis Java!", 60, 100);
    }
}
```

Comme on vient de le voir, lancer depuis Eclipse une application avec fenêtre, ne pose aucun problème – on crée une application qui ouvre une fenêtre et on l'exécute. AWT convient bien mais Swing est plus d'actualité.

Création d'applications Swing

La première application Swing est très simple : elle affiche le même message que l'applet et l'application AWT précédentes. Pour mettre en place cette application, on crée un projet `Ch06_03`, et dans le paquetage `org.eclipsebook.ch06` on ajoute la classe `Ch06_03` avec sa méthode `main`. Pour cet exemple, il faut utiliser les importations suivantes :

```
import java.awt.*;
import javax.swing.*;
import java.awt.event.*;
```

Dans cet exemple, on dérive une nouvelle classe `Panel` à partir de la classe `JPanel` et on y affiche le message :

```
class Panel extends JPanel
{
        .
        .
        .
}
```

La classe JPanel possède une méthode nommée paintComponent qu'il s'agit de surcharger afin d'afficher le texte du message de la façon suivante :

```
class Panel extends JPanel
{
        Panel()
        {
                setBackground(Color.white);
        }

        public void paintComponent (Graphics g)
        {
                super.paintComponent(g);
                g.drawString("Hello depuis Eclipse!", 60, 100);
        }
}
```

La principale classe de cette application, Ch06_03, dérive de la classe Swing JFrame :

```
public class Ch06_03 extends JFrame {
        .
        .
        .
```

Dans le constructeur de la classe Ch06_03, on crée l'objet de type Panel auquel on ajoute le panneau de l'objet JFrame :

```
Panel p;

public Ch06_03()
{
        super("Application Swing");

        Container contentPane = getContentPane();
        p = new Panel();
        contentPane.add(p);
}
```

Il reste à créer, dans main, une instance de la classe Ch06_03, d'afficher cet objet avec la méthode setVisible et de gérer la fermeture de la fenêtre avec la classe WindowAdapter comme on l'a fait dans AWT. On peut observer dans l'exemple 6-3 comment tout cela s'organise dans la méthode main.

Exemple 6-3. Un exemple simple d'application Swing

```
package org.eclipsebook.ch06;

import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class Ch06_03 extends JFrame {

        Panel p;

        public Ch06_03()
        {
```

Exemple 6-3. Un exemple simple d'application Swing (suite)

```
                super("Application Swing");

                Container contentPane = getContentPane();
                p = new Panel();
                contentPane.add(p);
        }

        public static void main(String args[])
        {
                final JFrame f = new Ch06_03();

                f.setBounds(100, 100, 300, 300);
                f.setVisible(true);
                f.setDefaultCloseOperation(DISPOSE_ON_CLOSE);

                f.addWindowListener(new WindowAdapter() {
                        public void windowClosing(WindowEvent e) {
                                System.exit(0);
                        }
                });
        }
}

class Panel extends JPanel
{
        Panel()
        {
                setBackground(Color.white);
        }

        public void paintComponent (Graphics g)
        {
                super.paintComponent(g);
                g.drawString("Hello depuis Eclipse!", 60, 100);
        }
}
```

L'exécution de cette application dans une fenêtre Swing est présentée dans la figure 6-4.

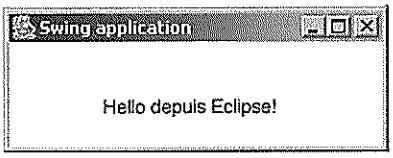


Figure 6-4. Une application Swing

L'intérêt majeur de Swing est de permettre au développeur de concevoir un *look-and-feel* (aspect intuitif) des applications compatible avec divers systèmes d'exploitation. A ce propos, un des atouts importants d'Eclipse est d'inclure le Standard Widget Toolkit

(SWT) d'IBM, une boîte à outils remplaçant AWT et Swing. Les deux chapitres qui suivent portent sur SWT, mais avant cela, on peut observer sur un autre exemple Ch06_04, les genres de look-and-feel que Swing peut offrir.

Pour donner un look-and-feel Swing à un programme, on utilise la méthode setLookAndFeel de la classe UIManager. Dans cet exemple, on place trois boutons radio pour utiliser les look-and-feel de Sun, Motif et Windows. Pour commencer, on dérive la classe Ch06_04 de la classe Swing JFrame et l'on crée les boutons radio :

```
public class Ch06_04 extends JFrame {
    JRadioButton b1 = new JRadioButton("Metal"),
    b2 = new JRadioButton("Motif"),
    b3 = new JRadioButton("Windows");
```

Comme dans l'exemple précédent, on utilise une instance de JPanel pour réaliser les affichages. Ici, on affiche un assortiment de contrôles, boutons, champs textes modifiables, cases à cocher, etc. et l'on ajoute le gestionnaire ActionListener aux boutons radio :

```
class Panel extends JPanel implements ActionListener
{
    public Panel()
    {
        add(new JButton("JButton"));
        add(new JTextField("JTextField"));
        add(new JCheckBox("JCheckBox"));
        add(new JRadioButton("JRadioButton"));
        add(new JLabel("JLabel"));
        add(new JList(new String[] {
            "JList Item 1", "JList Item 2", "JList Item 3"}));
        add(new JScrollBar(SwingConstants.HORIZONTAL));

        ButtonGroup group = new ButtonGroup();
        group.add(b1);
        group.add(b2);
        group.add(b3);

        b1.addActionListener(this);
        b2.addActionListener(this);
        b3.addActionListener(this);

        add(b1);
        add(b2);
        add(b3);
        .
        .
        .
```

On termine avec les appels de la méthode setLookAndFeel de la classe UIManager, afin de commuter sur les différents aspects intuitifs choisis. On peut voir comment tout cela fonctionne dans le listing complet de l'exemple 6-4.

Exemple 6-4. L'application Swing Ch06_04 avec les modules look-and-feel

```
package org.eclipsebook.ch06;
```

Exemple 6-4. L'application Swing Ch06_04 avec les modules look-and-feel (suite)

```
import java.awt.*;
import javax.swing.*;
import java.awt.event.*;

public class Ch06_04 extends JFrame {
    JRadioButton b1 = new JRadioButton("Metal"),
    b2 = new JRadioButton("Motif"),
    b3 = new JRadioButton("Windows");

    public Ch06_04() {
        super("Application Swing");

        Container contentPane = getContentPane();
        contentPane.add(new Panel(), BorderLayout.CENTER);
    }

    public static void main(String args[])
    {
        final JFrame f = new Ch06_04();

        f.setBounds(100, 100, 300, 300);
        f.setVisible(true);
        f.setDefaultCloseOperation(DISPOSE_ON_CLOSE);

        f.addWindowListener(new WindowAdapter() {
            public void windowClosing(WindowEvent e) {
                System.exit(0);
            }
        });
    }

    class Panel extends JPanel implements ActionListener
    {
        public Panel()
        {
            add(new JButton("JButton"));
            add(new JTextField("JTextField"));
            add(new JCheckBox("JCheckBox"));
            add(new JRadioButton("JRadioButton"));
            add(new JLabel("JLabel"));
            add(new JList(new String[] {
                "JList Item 1", "JList Item 2", "JList Item 3"}));
            add(new JScrollBar(SwingConstants.HORIZONTAL));

            ButtonGroup group = new ButtonGroup();
            group.add(b1);
            group.add(b2);
            group.add(b3);

            b1.addActionListener(this);
            b2.addActionListener(this);
```

Exemple 6-4. L'application Swing Ch06_04 avec les modules look-and-feel (suite)

```
            b3.addActionListener(this);

            add(b1);
            add(b2);
            add(b3);
        }

        public void actionPerformed(ActionEvent e)
        {
            JRadioButton src = (JRadioButton)e.getSource();

            try {
                if((JRadioButton)e.getSource() == b1)
                    UIManager.setLookAndFeel(
                      "javax.swing.plaf.metal.MetalLookAndFeel");
                else if((JRadioButton)e.getSource() == b2)
                    UIManager.setLookAndFeel(
                        "com.sun.java.swing.plaf.motif.MotifLookAndFeel");
                else if((JRadioButton)e.getSource() == b3)
                    UIManager.setLookAndFeel(
                        "com.sun.java.swing.plaf.windows.WindowsLookAndFeel");
            }
            catch(Exception ex) {}

            SwingUtilities.updateComponentTreeUI(getContentPane());
        }
    }
}
```

A l'exécution, on voit les divers contrôles installés dans la fenêtre, comme le montre la figure 6-5. Cette figure présente l'aspect *Metal* développée par Sun, qui est l'allure par défaut sur toute plateforme.

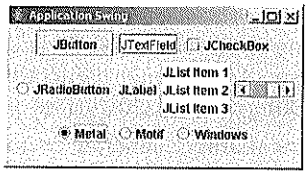


Figure 6-5. L'aspect Metal

Autrement, on change d'aspect intuitif en sélectionnant le bouton radio Motif, comme le montre la figure 6-6.

Enfin, en sélectionnant le bouton radio Windows, on affiche l'aspect Windows, comme on le voit sur la figure 6-7.

Comme on vient de le constater, les développeurs de Swing ont tenté de « coller » aux interfaces graphiques des différents systèmes d'exploitation. Malheureusement, les systèmes d'exploitation que Swing émule ont changé plus rapidement et aujourd'hui Swing est plutôt en retard, particulièrement sous Windows.

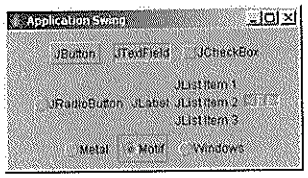


Figure 6-6. L'aspect Motif

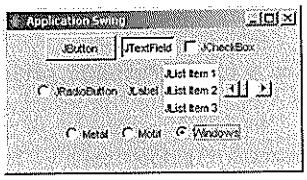


Figure 6-7. L'aspect Windows

Ce sujet est largement abordé dans le chapitre suivant, en commençant la présentation de SWT.

Maintenant que l'on sait concevoir les interfaces graphiques, on aimerait qu'Eclipse possède des outils visuels permettant d'installer les contrôles par simple glisser-déposer. Beaucoup de développeurs l'ont réclamé, mais il ne semble pas qu'Eclipse inclue rapidement une boîte à outils pour les interfaces graphiques basée sur le glisser-déposer. Cependant, il existe de nombreux modules Eclipse qui offrent cette fonctionnalité – certains sont commerciaux, d'autres gratuits. Dans la suite, on s'intéresse à un module gratuit qui permet la création d'applications Swing.

Utilisation de modules Eclipse

De nombreux modules Eclipse sont disponibles, dont plusieurs sont gratuits. On peut trouver plus de 400 modules sur *http://www.eclipse-plugins.2y.net/eclipse/*, et le module que l'on va utiliser pour Swing s'y trouve aussi. Cela étant, les modules Eclipse évoluant assez rapidement, on ne peut en donner les détails dans un ouvrage – le module à télécharger est sans doute différent de celui utilisé pour écrire ce livre.

Cela dit, on peut observer d'une manière générale ce qu'offre un module ; c'est ce que l'on se propose de faire ici. Dans ce chapitre on s'intéresse à un module gratuit assez connu, V4ALL, qui offre une aide pour l'écriture de code Swing ; il est disponible à l'adresse *http://www.eclipse-plugins.2y.net/eclipse/*. Il faut d'abord cliquer sur le lien *categories*, puis sur le lien *UI*, et enfin sur V4ALL Swing & SWT. Il faut alors arrêter Eclipse, puis procéder au téléchargement du fichier plug-in compressé dans le répertoire *eclipse* (c'est un répertoire qui contient les répertoires pour l'*environnement de travail* et les *modules*).

Le téléchargement des modules se fait normalement dans le répertoire *eclipse*. Ces fichiers sont compressés (zip ou tar) et habituellement on les décompresse dans le répertoire *eclipse*. Le fait de les décompresser ici force automatiquement le stockage des fichiers des modules dans les répertoires *plugins* et *features*, qui sont des sous-répertoires du répertoire *eclipse*.

Les plug-in ne sont pas toujours préparés pour être décompressés dans le répertoire *eclipse* – le lecteur consultera la documentation des modules avant toute chose. S'il s'agit d'un fichier à déziper, on doit ouvrir l'utilitaire unzip et regarder s'il faut déziper les fichiers dans le répertoire *plugins*. Si c'est le cas, il faut effectivement décompresser dans le répertoire *eclipse*. Cependant, certains modules se décompressent directement dans le répertoire *plugins*.

A présent, on peut décompresser (unzip ou untar) le module V4ALL, lequel doit automatiquement charger ses fichiers dans les répertoires *plugins* et *features*. C'est tout ce qu'il faut faire pour cette installation. On peut alors redémarrer Eclipse. Eclipse peut afficher une boîte de dialogue indiquant que la configuration a changé et peut demander à redémarrer Eclipse. Dans ce cas il faut encore redémarrer Eclipse.

Maintenant que le module V4ALL est installé, il est temps de s'en servir.

Utilisation du module V4ALL

Le module V4ALL possède une palette de composants Swing que l'on peut déposer sur la feuille à dessin pour construire son application. Pour ce faire, on crée une application Swing utilisant ce module. On crée un nouveau projet Java nommé `Ch06_05`, ainsi qu'un dossier source *src*, dans le troisième panneau de la boîte de dialogue Nouveau Projet, comme cela a déjà été fait. Enfin, on crée la classe `Ch06_05` dans le paquetage `org.eclipsebook.ch06`.

Ajout de l'éditeur V4ALL au projet

On ajoute un nouvel éditeur V4ALL au projet au moyen d'un clic droit sur le paquetage `org.eclipsebook.ch06`, puis l'on sélectionne Nouveau → Autre. La boîte de dialogue Nouveau s'ouvre comme sur la figure 6-8.

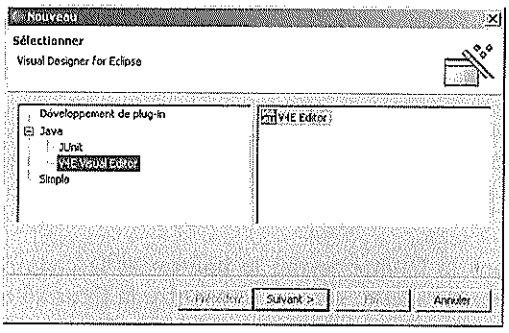


Figure 6-8. La boîte de dialogue Nouveau

On doit y voir l'option V4E Visual Editor ; on sélectionne cet item et l'on clique sur Suivant pour ouvrir la boîte de dialogue V4E que l'on voit dans la figure 6-9.

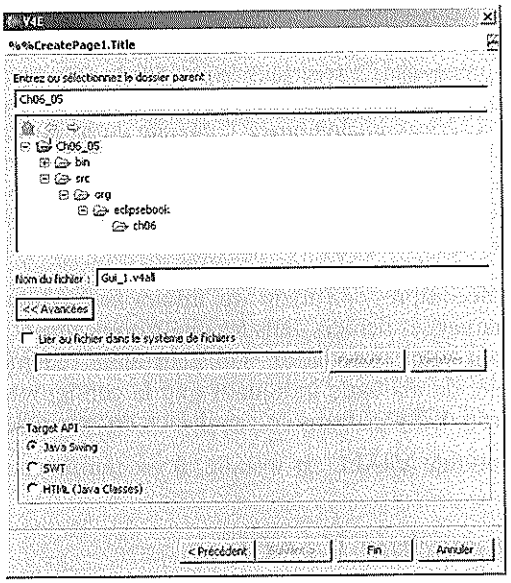


Figure 6-9. La boîte de dialogue V4E

La boîte de dialogue V4E permet de choisir le nom du fichier V4ALL qui contiendra toutes les informations de la conception V4ALL. On accepte le nom par défaut *Gui_1.v4all* et l'on clique sur Fin.

Conception d'une interface graphique

Après avoir donné un nom au fichier V4ALL, on clique sur Fin pour générer une perspective V4ALL, que l'on peut voir dans la figure 6-10. On notera que dans la Vue Packages, à gauche, on trouve une nouvelle entrée *Gui_1.v4all*. Au centre se trouve la palette de V4ALL à partir de laquelle on peut saisir les containers et les composants Swing. On crée l'application sur la feuille de dessin située à la droite de la palette. Sous la palette, la vue des propriétés présente les caractéristiques des éléments déposés sur la feuille à dessin.

Cet exemple Swing va afficher, dans une fenêtre Swing, un bouton et un champ texte modifiable ; pour cela, il faut chercher, dans la section Containers de la palette, l'option `Frame` et le déposer sur la feuille à dessin. On peut voir le résultat dans la figure 6-10.

Ensuite, on place dans la fenêtre Swing le bouton et le champ texte à partir de la section Component de la palette, comme le montre la figure 6-11 (noter que la fenêtre a été redimensionnée afin de mieux contenir les deux contrôles).

La légende par défaut est « Button 100 » ; pour la changer, on sélectionne le bouton et l'on clique sur la propriété `Text` dans la vue Propriétés. On clique sur le bouton « . . . » qui apparaît sur la ligne de propriété `Text`, puis l'on écrit « Cliquer Moi » dans le dialogue de saisie du texte qui apparaît et enfin OK.

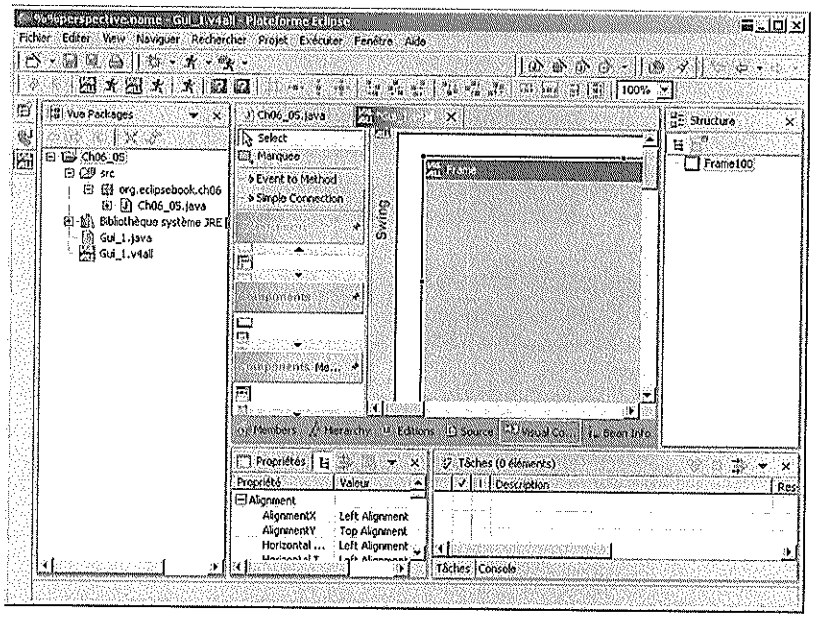


Figure 6-10. Utilisation de V4ALL

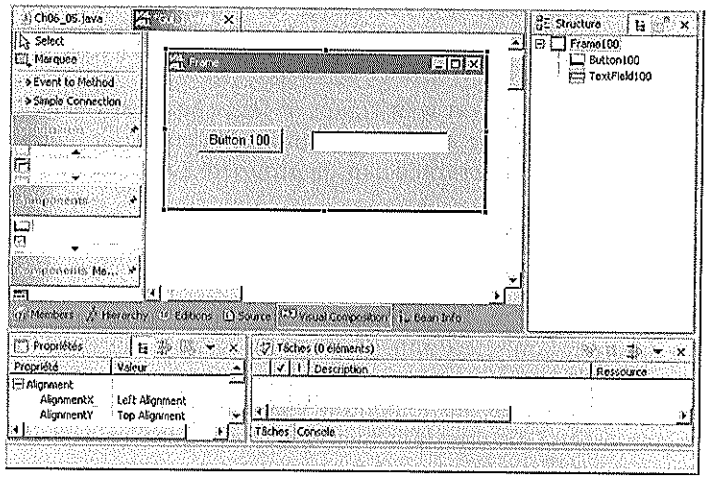


Figure 6-11. Ajout d'un bouton et d'un champ texte

Création d'une Méthode

L'objectif de cette application est d'afficher un message dans le champ texte quand l'utilisateur clique sur le bouton ; la prochaine étape est donc de créer la méthode qui traite le clic sur le bouton. Pour créer une nouvelle méthode, on dépose dans la feuille de dessin l'item `method` de la section Beans & Factories de la palette (l'item `method` est un icône contenant une paire de parenthèses). On double-clique l'item de la nouvelle méthode et l'on écrit son nom, `processEvent()`, comme on le voit dans la figure 6-12.

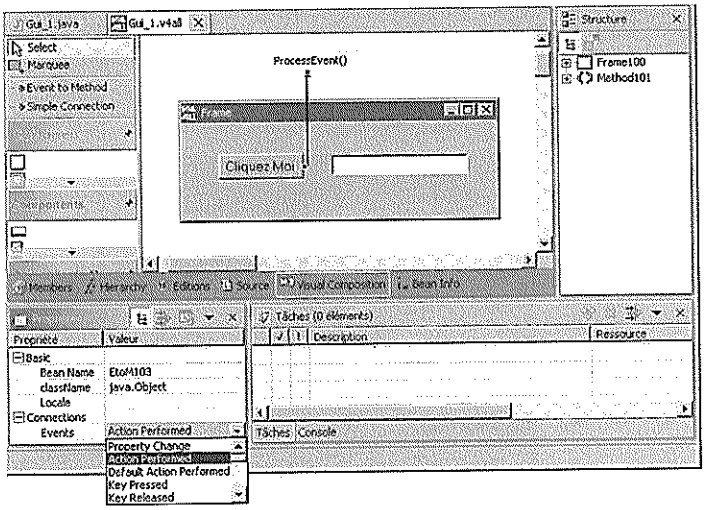


Figure 6-12. Création d'une nouvelle méthode

Pour attacher le bouton Cliquer Moi à la méthode, on sélectionne dans la palette l'item `Event to Method`, puis l'on clique sur le bouton. Ensuite, on amène la souris sur l'item de la méthode `processEvent` et l'on clique sur cet item. Cela relie visuellement, au moyen d'une flèche, le bouton et la méthode, comme on le voit dans la figure 6-12.

Pour indiquer quel événement on veut traiter, il faut cliquer sur Select, en haut de la palette et cliquer sur la flèche qui va du bouton à l'item de la méthode. Ensuite, on sélectionne l'item `Action Performed` dans la propriété `Events` de la vue Propriétés. C'est tout.

Génération et édition du code

Le code peut enfin être généré. Pour l'obtenir, on clique le bouton droit de la souris sur la feuille à dessin et l'on sélectionne l'option du menu contextuel Génération de code → Génération de code pour Swing (ou l'on clique dans la deuxième barre d'outils sous la barre de menu, sur le bouton V4ALL le plus à gauche, comme on le voit dans la figure 6-12). Cela crée le fichier *Gui_1.java*, que l'on voit ouvert dans la figure 6-13.

Ce code a été généré pour le programmeur par V4ALL. La méthode `processEvent` a été créée et sera appelée par un clic sur le bouton Cliquez-Moi. Si l'on ouvre la méthode `processEvent` on voit ce code généré :

```
public void processEvent(){
    // user code begin {1} Swing
    // user code end
    // user code begin {1} SWT
    // user code end
    // user code begin {1} HTML
    // user code end
    // user code begin {1} Eclipse
```

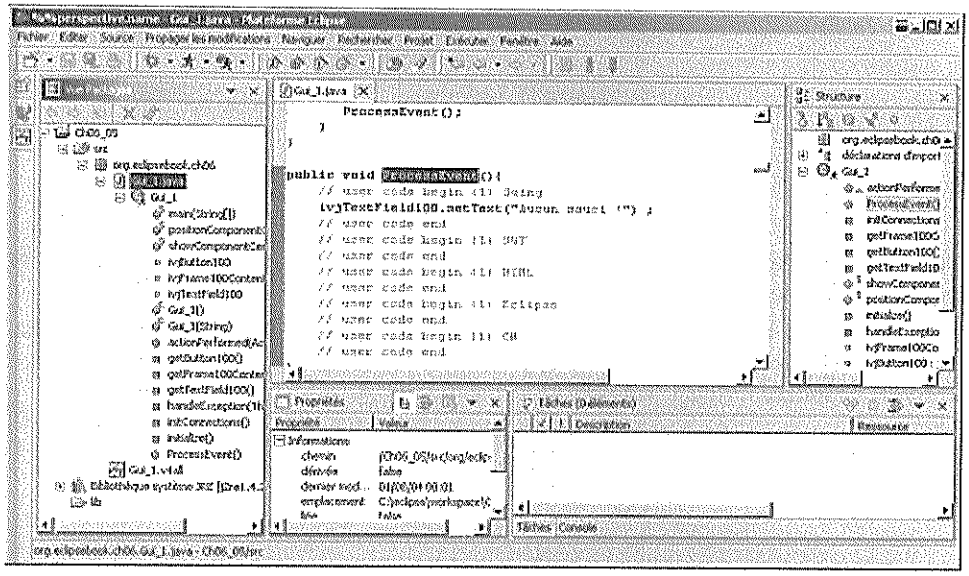


Figure 6-13. Génération de code au moyen de V4ALL

```
        // user code end
        // user code begin {1} C#
        // user code end
    }
```

Comme on peut le constater, le champ texte est référencé par `ivjTextField100` et l'on utilise la méthode d'instance `setText` pour y afficher le message « Pas de souci! » (on peut voir le code saisi dans la figure 6-13) :

```
    public void processEvent(){
        // user code begin {1} Swing
        ivjTextField100.setText("Pas de souci!");
        // user code end
        // user code begin {1} SWT
        // user code end
        // user code begin {1} HTML
        // user code end
        // user code begin {1} Eclipse
        // user code end
        // user code begin {1} C#
        // user code end
    }
```

Exécution du résultat

Il n'y a rien d'autre à faire. À présent, on enregistre le fichier *Gui_1.java*, puis on le sélectionne dans la Vue Packages et on ouvre l'option de menu Exécuter en tant que → Application Java, lançant ainsi cette nouvelle application Swing, comme le montre la figure 6-14. Lorsqu'on clique sur le bouton, le champ texte présente le message voulu.

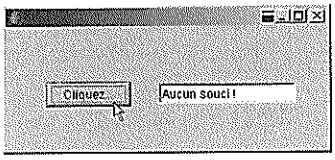


Figure 6-14. La nouvelle application Swing

Et voilà, tout marche comme prévu. Une application Swing a été conçue et exécutée grâce au module additionnel V4ALL.

Nous avons fait le tour de V4ALL ainsi que des applets AWT et Swing. Dans le chapitre suivant, nous aborderons une alternative pour les GUI qui fait partie d'Eclipse : le SWT (Standard Widget Toolkit).

CHAPITRE 7

SWT : boutons, textes, labels, listes, gestionnaires de placement et evénements

Le SWT (Standard Widget Toolkit), est un ensemble d'outils créé par IBM pour remplacer l'AWT (Abstract Windowing Toolkit) et la bibliothèque Swing de Java. Sa première qualité est d'être intégré à Eclipse et immédiatement prêt à l'emploi. Voici comment est présenté SWT dans la documentation d'Eclipse :

> Le Standard Widget Toolkit (SWT) est une boîte à outils d'objets élémentaires (widgets) pour le développeur Java ; le SWT propose une API portable reposant sur les couches basses du GUI (interface graphique utilisateur) de l'OS natif.

Graphiques Java

Les travaux sur le graphisme en Java ont déjà une longue et glorieuse histoire et l'on est en droit d'espérer un avenir aussi radieux. Cette histoire a commencé avec le très basique AWT, s'est poursuivie avec la montée en puissance due au paquetage Swing et connaît aujourd'hui une progression très significative avec l'arrivée de SWT.

AWT

AWT, première occurrence de boîte à outils Java pour les GUI (interfaces graphiques utilisateur), a été écrite en quelques semaines. Elle permet au développeur d'afficher des fenêtres et divers contrôles tels que des zones de texte et des boutons. Les interfaces graphiques AWT sont très faciles à développer et elles utilisent les contrôles du système d'exploitation sous-jacent – par exemple, sous Windows, une zone de texte est une zone de texte Windows. Sous Mac, on obtient une zone de texte Mac. Chaque système d'exploitation possède son propre jeu de contrôles : c'est la raison pour laquelle Sun a seulement implémenté les contrôles communs à tous ces systèmes d'exploitation, limitant ainsi AWT à un ensemble de contrôles relativement simples.

Swing

Afin de satisfaire les développeurs, toujours plus exigeants, Java s'est doté de Swing, qui offre une implémentation non native de contrôles sophistiqués tels que des arborescences, des tables et du texte. Cet accroissement significatif des possibilités a eu pour corollaire des applications développées avec Swing qui ne sont pas standard et sont très spécifiques de Java.

Sun a ajouté l'émulation de « look-and-feel » (aspects d'une interface intuitive qui ont été vus au chapitre précédent) afin de permettre aux applications d'adopter l'aspect propre au système d'exploitation sur lequel elles s'exécutent, mais il n'a pas été possible de suivre les évolutions des systèmes d'exploitation (tels que Windows ME, 2000, XP et autres). En plus, ces GUI étant implémentés en Java et non pas en natif, ils présentent un temps de réponse assez détestable par rapport à des applications écrites en code natif.

SWT

Sur tous ces sujets, SWT est pertinent car il offre un ensemble d'éléments graphiques construits à partir des contrôles natifs (via l'interface JNI – Java Native Interface), quand bien sûr ces contrôles existent. Sinon, SWT crée ses propres contrôles pour compenser les déficiences du systèmes d'exploitation. Cela signifie que du code natif est nécessaire pour être adapté à chaque système d'exploitation, mais jusqu'à présent, IBM a été en mesure d'en maintenir les mises à jour. Enfin, SWT est intégré à Eclipse, caractéristique particulièrement intéressante.

SWT remplace AWT et Swing en apportant un enrichissement subtantiel du contenu et des caractéristiques. Ce chapitre et le suivant ne sont qu'une introduction à SWT et n'offrent qu'un aperçu de ce qu'il faut connaître de SWT pour concevoir des applications conviviales et sophistiquées avec SWT. Nous ferons ainsi le point sur ce que sont les widgets (les éléments de l'interface utilisateur que l'on utilise avec SWT), les layouts (gestionnaires de placement des éléments dans l'interface graphique) et les événements. De nombreux exemples exécutables illustrent cette présentation et permettent de rendre compte du fonctionnement de SWT.

SWT est porté sur un nombre toujours plus important d'environnements graphiques. Le portage est actuellement réalisé sur Windows, Linux GTK, Linux Motif, Solaris Motif, AIX Motif, HPUX Motif, Photon QNX, et Mac OS X.

Un exemple SWT

Plutôt qu'une présentation abstraite de SWT, nous avons choisi une approche concrète et pratique de sa découverte. En codant un exemple, on perçoit mieux l'histoire de SWT et ce qu'il faut rassembler pour une application SWT. Le premier exemple est très simple, il affiche le message « Pas de souci! » dans une fenêtre SWT.

Dans cet objectif, on crée un nouveau projet, `Ch07_01`, et on lui ajoute la classe `Ch07_01` que l'on place dans le paquetage `org.eclipsebook.ch07`. Pour pouvoir travailler avec SWT, il faut typiquement commencer avec les deux consignes d'importation suivantes :

```
import org.eclipse.swt.widgets.*;
import org.eclipse.swt.*;
```

Pour que ces deux importations aient un effet, il est nécessaire d'inclure *swt.jar* dans le chemin de compilation. Il convient de se souvenir que SWT dépend du système d'exploitation : il y a donc un fichier *swt.jar* par système d'exploitation. Pour ajouter

swt.jar au projet `Ch07_01`, il suffit de sélectionner ce projet dans la Vue Packages, et avec un clic droit de sélectionner Propriétés. Dans la boîte de dialogue Propriétés de `Ch07_01`, on sélectionne l'option Chemin de compilation Java et l'on clique sur le bouton Ajouter des fichiers JAR externes. On navigue ensuite dans le système de fichiers pour atteindre *swt.jar*, lequel se trouve, selon le système d'exploitation utilisé, dans l'un des répertoires suivants (noter que *INSTALLDIR* est le répertoire d'installation d'Eclipse et que le chemin dépend de la version actuellement en service 2.1.1 à 2.1.3 ou tout autre valeur) :

Win32

INSTALLDIR\eclipse\plugins\org.eclipse.swt.win32_2.1.1\ws\win32\swt.jar

Linux GTK

INSTALLDIR/eclipse/plugins/org.eclipse.swt.gtk_2.1.1/ws/gtk/swt.jar

Linux Motif

INSTALLDIR/eclipse/plugins/org.eclipse.swt.motif_2.1.1/ws/motif/swt.jar

Solaris Motif

INSTALLDIR/eclipse/plugins/org.eclipse.swt.motif_2.1.1/ws/solaris/sparc/swt.jar

AIX Motif

INSTALLDIR/eclipse/plugins/org.eclipse.swt.motif_2.1.1/ws/aix/ppc/swt.jar

HPUX Motif

INSTALLDIR/eclipse/plugins/org.eclipse.swt.motif_2.1.1/ws/hpux/PA_RISC/swt.jar

Photon QNX

INSTALLDIR/eclipse/plugins/org.eclipse.swt.photon_2.1.1/ws/photon/swt.jar

Mac OS X

INSTALLDIR/eclipse/plugins/org.eclipse.swt.carbon_2.1.1/ws/carbon/swt.jar

Après avoir localisé *swt.jar*, il suffit de cliquer sur Ouvrir puis sur OK pour l'ajouter au chemin de compilation de ce projet.

Certains systèmes d'exploitation, comme Linux GTK, nécessitent un ou plusieurs fichiers JAR pour pouvoir exécuter SWT (dans Linux GTK, il faut utiliser *swt.jar* et *swt-pi.jar*). Dans ce cas, il faut ajouter tous les fichiers JAR au chemin de compilation. (Tous les fichiers requis devant être dans le même répertoire.)

Au niveau du codage, la prochaine étape consiste à créer un objet `Display`, lequel représente une *session SWT*. Cet objet constitue une connexion entre SWT et le GUI (interface graphique utilisateur) du système d'exploitation. On utilise cet objet `Display` pour démarrer la boucle de contrôle des événements et la communication entre le thread principal de l'interface utilisateur et les autres threads. Voici comment est créé l'objet `Display` dans la méthode `main` de la classe `Ch07_01` :

```
import org.eclipse.swt.widgets.*;
import org.eclipse.swt.*;
```

```
public class Ch07_01 {

        public static void main(String [] args) {
           Display display = new Display();
        .
        .
        .
}
```

Puisqu'on dispose à présent d'un objet `Display`, c'est à travers les objets `Shell` que l'on peut créer autant de fenêtres que l'on veut. Dans SWT, un shell est une fenêtre gérée par le système de fenêtrage du système d'exploitation. Au sommet de la hiérarchie, il y a un shell, fils du display, qui est une fenêtre que l'utilisateur peut redimensionner, déplacer, minimiser, etc. On peut créer des shells secondaires, enfants d'autres shells, tels que des boîtes de dialogue ou des boîtes de message. Voici comment on crée et on dimensionne le shell à utiliser – on notera que grâce au paramètre `display` que l'on transmet au constructeur du `Shell`, le shell est un fils de l'objet `display`:

```
import org.eclipse.swt.widgets.*;
import org.eclipse.swt.*;

public class Ch07_01 {

        public static void main(String [] args) {
           Display display = new Display();
           Shell shell = new Shell(display);
           shell.setSize(300, 200);
        .
        .
        .
        }
}
```

À présent, on peut compléter l'application au moyen de widgets (éléments de l'interface utilisateur). Dans la documentation de SWT, le terme *widget* est utilisé indifféremment pour désigner un *contrôle* ou une structure *composite*. La documentation SWT donne de *widget* une définition technique, formelle, et qui plus est récursive : « la classe abstraite de tout objet de l'interface utilisateur qui peut être placée à l'intérieur d'un autre widget ». Plus concrètement, le terme *widget* signifie en général tout élément SWT de l'interface utilisateur. Les *composites* sont des widgets conçus pour avoir des enfants, comme les barres de menu, les arborescences (trees) ou les feuilles de dessin (canvases). Les *contrôles* sont les widgets ayant un équivalent dans le système d'exploitation, comme les boutons, les listes et les étiquettes. Le tableau 7-1 présente tous les contrôles SWT. Elle présente en même temps les styles et les événements associés à chaque contrôle.

Tableau 7-1. Contrôles et options SWT

Widget	Fonction	Styles	Evénements
Button	Le bouton standard.	BORDER, CHECK, PUSH, RADIO, TOGGLE, FLAT, LEFT, RIGHT, CENTER, ARROW (with UP, DOWN)	Dispose, FocusIn, FocusOut, Help, KeyDown, KeyUp, MouseDoubleClick, MouseDown, MouseEnter, MouseExit, MouseHover, MouseUp, MouseMove, Move, Paint, Resize, Selection
Canvas	Contrôle composite capable de contenir d'autres contrôles, mais qui peut être aussi une feuille de dessin ; ce contrôle est souvent utilisé comme base pour des contrôles personnalisés.	BORDER, H_SCROLL, V_SCROLL, NO_BACKGROUND, NO_FOCUS, NO_MERGE_PAINTS, NO_REDRAW_RESIZE, NO_RADIO_GROUP	Dispose, FocusIn, FocusOut, Help, KeyDown, KeyUp, MouseDoubleClick, MouseDown, MouseEnter, MouseExit, MouseHover, MouseUp, MouseMove, Move, Paint, Resize
Caret	Le caret en forme de I qui indique le point d'insertion du texte.		Dispose
Combo	Le sélecteur d'options standard – qui est une combinaison de contrôle textuel avec une liste déroulante.	BORDER, DROP_DOWN, READ_ONLY, SIMPLE	Dispose, FocusIn, FocusOut, Help, KeyDown, KeyUp, MouseDoubleClick, MouseDown, MouseEnter, MouseExit, MouseHover, MouseUp, MouseMove, Move, Paint, Resize, DefaultSelection, Modify, Selection
Composite	Contrôle contenant d'autres éléments de l'interface grahique.	BORDER, H_SCROLL, V_SCROLL	Dispose, FocusIn, FocusOut, Help, KeyDown, KeyUp, MouseDoubleClick, MouseDown, MouseEnter, MouseExit, MouseHover, MouseUp, MouseMove, Move, Paint, Resize
CoolBar	Contrôle composite permettant à l'utilsateur de repositionner dynamiquement les items contenus.	BORDER	Dispose, FocusIn, FocusOut, Help, KeyDown, KeyUp, MouseDoubleClick, MouseDown, MouseEnter, MouseExit, MouseHover, MouseUp, MouseMove, Move, Paint, Resize
CoolItem	Elément d'interface utilisateur représentant une zone repositionnable dans un CoolBar.	DROP_DOWN	Dispose
Group	Contrôle composite regroupant tous ensembles d'autres éléments graphiques. Il peut les enlacer avec une bordure et posséder une légende.	BORDER, SHADOW_ETCHED_IN, SHADOW_ETCHED_OUT, SHADOW_IN, SHADOW_OUT, SHADOW_NONE	Dispose, FocusIn, FocusOut, Help, KeyDown, KeyUp, MouseDoubleClick, MouseDown, MouseEnter, MouseExit, MouseHover, MouseUp, MouseMove, Move, Paint, Resize
Label	La légende standard – affiche un texte ou une image.	BORDER, CENTER, LEFT, RIGHT, WRAP, SEPARATOR (with HORIZONTAL, SHADOW_IN, SHADOW_OUT, SHADOW_NONE, VERTICAL)	Dispose, FocusIn, FocusOut, Help, KeyDown, KeyUp, MouseDoubleClick, MouseDown, MouseEnter, MouseExit, MouseHover, MouseUp, MouseMove, Move, Paint, Resize
List	La liste standard – permet à l'utilisateur de choisir un élément dans une liste.	BORDER, H_SCROLL, V_SCROLL, SINGLE, MULTI	Dispose, FocusIn, FocusOut, Help, KeyDown, KeyUp, MouseDoubleClick, MouseDown, MouseEnter, MouseExit, MouseHover, MouseUp, MouseMove, Move, Paint, Resize, Selection, DefaultSelection
Menu	Le menu standard – contient les options de menu.	BAR, DROP_DOWN, NO_RADIO_GROUP, POP_UP	Dispose, Help, Hide, Show

Tableau 7-1. Contrôles et options SWT (suite)

Widget	Fonction	Styles	Evénements
MenuItem	Objet UI représentant une option de menu.	CHECK, CASCADE, PUSH, RADIO, SEPARATOR	Dispose, Arm, Help, Selection
ProgressBar	La barre de progression standard – indique à l'utilisateur l'état d'avancement d'une tâche, généralement sous la forme d'un bar graph.	BORDER, INDETERMINATE, SMOOTH, HORIZONTAL, VERTICAL	Dispose, FocusIn, FocusOut, Help, KeyDown, KeyUp, MouseDoubleClick, MouseDown, MouseEnter, MouseExit, MouseHover, MouseUp, MouseMove, Move, Paint, Resize
Sash	Contrôle permettant à l'utilisateur de séparer une fenêtre en volets et de redimensionner les éléments de l'interface en déplaçant la frontière élastique.	BORDER, HORIZONTAL, VERTICAL	Dispose, FocusIn, FocusOut, Help, KeyDown, KeyUp, MouseDoubleClick, MouseDown, MouseEnter, MouseExit, MouseHover, MouseUp, MouseMove, Move, Paint, Resize, Selection
Scale	Contrôle représentant une étendue de valeurs numériques.	BORDER, HORIZONTAL, VERTICAL	Dispose, FocusIn, FocusOut, Help, KeyDown, KeyUp, MouseDoubleClick, MouseDown, MouseEnter, MouseExit, MouseHover, MouseUp, MouseMove, Move, Paint, Resize, Selection
ScrollBar	Barre de défilement standard – représente une étendue de valeurs numériques positives.	HORIZONTAL, VERTICAL	Dispose, Selection
Shell	Fenêtre gérée par le système de fenêtrage du système d'exploitation.	BORDER, H_SCROLL, V_SCROLL, CLOSE, MIN, MAX, NO_TRIM, RESIZE, TITLE (voir aussi SHELL_TRIM,DIALOG_TRIM)	Dispose, FocusIn, FocusOut, Help, KeyDown, KeyUp, MouseDoubleClick, MouseDown, MouseEnter, MouseExit, MouseHover, MouseUp, MouseMove, Move, Paint, Resize, Activate, Close, Deactivate,Deiconify, Iconify
Slider	Contrôle représentant un ensemble de valeurs numériques; l'utilisateur en sélectionne une en déplaçant un index.	BORDER, HORIZONTAL, VERTICAL	Dispose, FocusIn, FocusOut, Help, KeyDown, KeyUp, MouseDoubleClick, MouseDown, MouseEnter, MouseExit, MouseHover, MouseUp, MouseMove, Move, Paint, Resize, Selection
TabFolder	Contrôle composite qui regroupe des contrôles en pages que l'utilisateur sélectionne au moyen d'onglets.	BORDER	Dispose, FocusIn, FocusOut, Help, KeyDown, KeyUp, MouseDoubleClick, MouseDown, MouseEnter, MouseExit, MouseHover, MouseUp, MouseMove, Move, Paint, Resize, Selection
TabItem	Contrôle correspondant à un onglet du TabFolder.		Dispose
Table	Contrôle affichant une liste de TableItem.	BORDER, H_SCROLL, V_SCROLL, SINGLE, MULTI, CHECK, FULL_SELECTION, HIDE_SELECTION	Dispose, FocusIn, FocusOut, Help, KeyDown, KeyUp, MouseDoubleClick, MouseDown, MouseEnter, MouseExit, MouseHover, MouseUp, MouseMove, Move, Paint, Resize, Selection, DefaultSelection
TableColumn	Objet UI représentant une colonne dans une table.	LEFT, RIGHT, CENTER	Dispose, Move, Resize, Selection
TableItem	Objet UI représentant un item dans une table.		Dispose

Tableau 7-1. Contrôles et options SWT (suite)

Widget	Fonction	Styles	Evénements
Text	Le contrôle standard du texte, qui permet à l'utilisateur d'éditer du texte.	BORDER, SINGLE, READ_ONLY, LEFT, CENTER, RIGHT, WRAP, MULTI (avec H_SCROLL, V_SCROLL)	Dispose, FocusIn, FocusOut, Help, KeyDown, KeyUp, MouseDoubleClick, MouseDown, MouseEnter, MouseExit, MouseHover, MouseUp, MouseMove, Move, Paint, Resize, DefaultSelection, Modify, Verify
ToolBar	La barre d'outils standard – contrôle composite qui contient les ToolItem d'une barre d'outils.	BORDER, FLAT, WRAP, RIGHT, SHADOW_OUT HORIZONTAL, VERTICAL	Dispose, FocusIn, FocusOut, Help, KeyDown, KeyUp, MouseDoubleClick, MouseDown, MouseEnter, MouseExit, MouseHover, MouseUp, MouseMove, Move, Paint, Resize
ToolItem	Objet UI qui représente un item de barre d'outils.	PUSH, CHECK, RADIO, SEPARATOR, DROP_DOWN	Dispose, Selection
Tracker	Objet UI qui implémente le rectangle élastique.	LEFT, RIGHT, UP, DOWN, RESIZE	Dispose, Move, Resize
Tree	Contrôle pour l'affichage d'une liste arborescente hiérarchisée de TreeItem.	BORDER, H_SCROLL, V_SCROLL, SINGLE, MULTI, CHECK	Dispose, FocusIn, FocusOut, Help, KeyDown, KeyUp, MouseDoubleClick, MouseDown, MouseEnter, MouseExit, MouseHover, MouseUp, MouseMove, Move, Paint, Resize, Selection, DefaultSelection, Collapse, Expand
TreeItem	Objet UI qui représente un item d'une liste arborescente.		Dispose

Dans l'exemple suivant, on affiche le message « Pas de souci! » en utilisant une étiquette. Dans le tableau 7-1 on présente les divers styles possibles de tous les contrôles SWT ; ici, on s'intéresse uniquement aux styles propres aux étiquettes :

`SWT.BORDER`
: Ajoute une bordure

`SWT.CENTER`
: Centre le texte

`SWT.LEFT`
: Réalise la justification à gauche du texte

`SWT.RIGHT`
: Réalise la justification à droite du texte

`SWT.WRAP`
: Passage à la ligne automatique sans césure des mots

`SWT.SEPARATOR`
: Marque une séparation dans le texte

Voici comment créer une étiquette avec le texte adéquat – noter que le texte est centré au moyen du style `SWT.CENTER` et que les limites correspondent à celle de la fenêtre d'accueil :

```
import org.eclipse.swt.widgets.*;
import org.eclipse.swt.*;
```

```
public class Ch07_01 {

        public static void main(String [] args) {
           Display display = new Display();
           Shell shell = new Shell(display);
           shell.setSize(300, 100);
           Label label = new Label(shell, SWT.CENTER);
           label.setText("Pas de souci!");
           label.setBounds(shell.getClientArea());
        .
        .
        .
}
```

Finalement, on ouvre la fenêtre (shell) pour l'afficher et l'on met en place une boucle de répartition des événements. Généralement, cette boucle tourne jusqu'à ce que l'utilisateur ferme la fenêtre principale. Dans le corps de la boucle, on vérifie si l'objet `display` doit dispenser des messages, sinon, en attendant, on place cet objet en sommeil. Voir la totalité du listing avec l'exemple 7-1.

Exemple 7-1. Première application SWT, Ch07_01.java

```
import org.eclipse.swt.widgets.*;
import org.eclipse.swt.*;

public class Ch07_01 {

        public static void main(String [] args) {
           Display display = new Display();
           Shell shell = new Shell(display);
           shell.setSize(300, 100);
           Label label = new Label(shell, SWT.CENTER);
           label.setText("Pas de souci!");
           label.setBounds(shell.getClientArea());
           shell.open();
           while(!shell.isDisposed()) {
                  if(!display.readAndDispatch()) display.sleep();
           }
           display.dispose();
        }
}
```

Il faut remarquer que lorsqu'on en a fini avec l'objet `display`, on le libère avec sa méthode `dispose`. La libération des ressources n'est pas nécessaire dans AWT et Swing ; dans le cas de SWT, en revanche, c'est le système d'exploitation qui alloue et libère les ressources de façon explicite. SWT exige donc du programmeur de libérer toutes les ressources du système d'exploitation qu'il a allouées ; à cette fin, il peut utiliser la méthode *`widget`*`.dispose`.

Le codage est terminé, mais cet exemple n'est pas en mesure de s'exécuter ; la prochaine étape consiste à ajouter le code natif de la librairie JNI dans le chemin de compilation, afin que la machine virtuelle Java puisse le trouver. Il convient de se souvenir que SWT utilise le code natif pour l'affichage, ce qui signifie qu'il faut faire appel à JNI. Pour ce faire, on sélectionne dans la Vue Packages la classe à exécuter

(ici `Ch07_01`) et on sélectionne l'option de menu Exécuter → Exécuter pour choisir la configuration d'exécution.

Dans la boîte de dialogue de configuration d'exécution, on sélectionne le nom de la classe sous l'item Application Java et l'on clique sur le bouton Nouveau. Le nom du projet et les champs de l'onglet Main sont déjà remplis – si ce n'est pas le cas, il faut les remplir. Dans la zone Arguments VM de l'onglet Arguments, on insère la localisation de la librairie SWT, laquelle dépend du système d'exploitation (ces chemins sont à corriger en fonction de la version d'Eclipse installée ; changer 2.1.1 en 2.1.3 ou autre si nécessaire) :

Win32

-Djava.library.path=INSTALLDIR\plugins\org.eclipse.swt.win32_2.1.1\os\win32\x86

Linux GTK

-Djava.library.path=INSTALLDIR/eclipse/plugins/org.eclipse.swt.gtk_2.1.1/os/linux/x86

Linux Motif

-Djava.library.path=INSTALLDIR/eclipse/plugins/org.eclipse.swt.motif_2.1.1/os/linux/x86

Solaris Motif

-Djava.library.path=INSTALLDIR/eclipse/plugins/org.eclipse.swt.motif_2.1.1/os/solaris/sparc

AIX Motif

-Djava.library.path=INSTALLDIR/eclipse/plugins/org.eclipse.swt.motif_2.1.1/os/aix/ppc

HPUX Motif

-Djava.library.path=INSTALLDIR/eclipse/plugins/org.eclipse.swt.motif_2.1.1/os/hpux/PA_RISC

Photon QNX

-Djava.library.path=INSTALLDIR/eclipse/plugins/org.eclipse.swt.photon_2.1.1/os/qnx/x86

Mac OS X

-Djava.library.path=INSTALLDIR/eclipse/plugins/org.eclipse.swt.carbon_2.1.1/os/macosx/ppc

On clique successivement sur les boutons Appliquer puis Exécuter. Le résultat de cette exécution apparaît dans la figure 7-1.

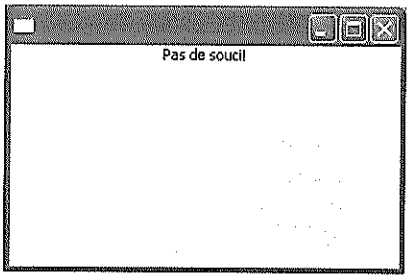


Figure 7-1. La première application SWT

Voici donc une première application SWT – certes pas très compliquée, mais qui nous a permis d'aborder les rudiments des applications SWT.

Utilisation des boutons

Le nouvel objectif est d'ajouter à l'application SWT des éléments d'interface interactifs. Dans l'exemple qui suit, un clic sur un bouton provoque l'affichage d'un texte dans un contrôle textuel. On démarre de la même façon mais, cette fois, on ajoute les importations `org.eclipse.swt.events.*` pour traiter les clics sur le bouton, et `org.eclipse.swt.layout.*` pour choisir une politique de placement des contrôles. Afin d'enrichir cet exemple, nous donnons un titre à la fenêtre, grâce à la méthode `setText` du shell :

```
package org.eclipsebook.ch07;

import org.eclipse.swt.widgets.*;
import org.eclipse.swt.SWT;
import org.eclipse.swt.events.*;
import org.eclipse.swt.layout.*;

        public static void main(String [] args) {
           Display display = new Display();
           Shell shell = new Shell(display);
           shell.setSize(300, 100);
           shell.setLayout(new RowLayout());
           shell.setText("Exemple de bouton");
        .
        .
        .
```

À présent, on dispose plusieurs contrôles et l'on choisit, comme politique de placement SWT, le placement *row layout* qui affiche les contrôles en ligne :

```
public class Ch07_02 {

        public static void main(String [] args) {
           Display display = new Display();
           Shell shell = new Shell(display);
           shell.setSize(300, 100);
           shell.setText("Exemple de bouton");
           shell.setLayout(new RowLayout());
        .
        .
        .
```

La disposition en ligne n'est qu'une des nombreuses possibilités de placement des contrôles ; on en verra d'autres dans la prochaine section. Pour le moment, on ajoute un bouton grâce à la classe `Button` et un contrôle texte avec la classe `Text`. Il existe différents styles pour les boutons :

`SWT.BORDER`
: Ajoute une bordure

`SWT.CHECK`
Crée une case à cocher

`SWT.PUSH`
Crée un bouton poussoir

`SWT.RADIO`
Crée un bouton radio

`SWT.TOGGLE`
Crée un bouton à bascule (commutateur)

`SWT.FLAT`
Crée un bouton plat

`SWT.LEFT`
Choisit un alignement à gauche

`SWT.RIGHT`
Choisit un alignement à droite

`SWT.CENTER`
Choisit un alignement au centre

`SWT.ARROW`
Crée un bouton avec un symbole tête de flèche

Pour le contrôle texte, les styles possibles sont:

`SWT.BORDER`
Ajoute une bordure

`SWT.SINGLE`
Autorise une sélection unique

`SWT.READ_ONLY`
Limite le texte à la lecture uniquement

`SWT.LEFT`
Choisit l'alignement à gauche

`SWT.CENTER`
Choisit l'alignement au centre

`SWT.RIGHT`
Choisit l'alignement à droite

`SWT.WRAP`
Autorise le passage à la ligne automatique

`SWT.MULTI`
Autorise les sélections multiples

Voici comment on place le bouton et le texte:

```
public class Ch07_02 {

        public static void main(String [] args) {
           Display display = new Display();
           Shell shell = new Shell(display);
           shell.setSize(300, 100);
           shell.setText("Exemple de bouton");
           shell.setLayout(new RowLayout());

           final Button bouton = new Button(shell, SWT.PUSH);
            bouton.setText("Cliquez moi");

           final Text texte = new Text(shell, SWT.SHADOW_IN);
        .
        .
        .
```

Pour traiter le clic sur le bouton, on lui ajoute un `SelectionListener` grâce à la méthode `addSelectionListener`. Les récepteurs d'événements (listeners) de SWT sont très proches de ceux de AWT ; en voici quelques-uns parmi les plus usités :

`ControlListener`
: Traite les mouvements et les redimensionnements

`FocusListener`
: Traite le passage au premier plan ou à l'arrière plan (focus)

`KeyListener`
: Traite la frappe au clavier

`MouseListener`, `MouseMoveListener`, `MouseTrackListener`
: Gèrent la souris

`SelectionListener`
: Traite la sélection d'un élément (y compris le clic d'un bouton)

L'interface `SelectionListener` possède deux méthodes qu'il faut implémenter : `widgetSelected` lorsque le contrôle est sélectionné et `widgetDefaultSelected` lorsqu'une sélection par défaut est donnée au contrôle. Ici, on affiche « Pas de souci ! » dans le contrôle texte, grâce à une classe interne anonyme :

```
button.addSelectionListener(new SelectionListener()
{
        public void widgetSelected(SelectionEvent event)
        {
                texte.setText("Pas de souci !");
        }

        public void widgetDefaultSelected(SelectionEvent event)
        {
                texte.setText("Pas de souci !");
        }
});
```

Tout ce qui reste à faire, c'est afficher la fenêtre et implémenter la boucle de capture des événements comme le montre l'exemple 7-2.

Exemple 7-2. Utilisation des boutons SWT

```
package org.eclipsebook.ch07;

import org.eclipse.swt.widgets.*;
import org.eclipse.swt.SWT;
import org.eclipse.swt.events.*;
import org.eclipse.swt.layout.*;

public class Ch07_02 {

        public static void main(String [] args) {
           Display display = new Display();
           Shell shell = new Shell(display);
           shell.setSize(300, 100);
           shell.setLayout(new RowLayout());
           shell.setText("Exemple de bouton");

           final Button bouton = new Button(shell, SWT.PUSH);
        bouton.setText("Cliquez moi");

           final Text texte = new Text(shell, SWT.SHADOW_IN);

           bouton.addSelectionListener(new SelectionListener()
           {

                 public void widgetSelected(SelectionEvent event)
                 {
                       texte.setText("Pas de souci !");
                 }

                 public void widgetDefaultSelected(SelectionEvent event)
                 {
                       texte.setText("Pas de souci !");
                 }
           });

           shell.open();
           while(!shell.isDisposed()) {
                 if(!display.readAndDispatch()) display.sleep();
           }
           display.dispose();
        }
}
```

Le résultat est visible dans la figure 7-2 ; quand on appuie sur le bouton, le message apparaît dans le contrôle texte.

Dans l'exemple précédent, nous avons utilisé une disposition en ligne, mais on peut fixer directement les coordonnées des contrôles avec la méthode `setBounds(`*x*, *y*, *width*, *height*`)`. Par exemple, si l'on veut fixer explicitement les positions et les tailles

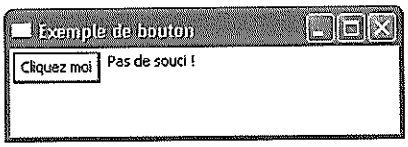


Figure 7-2. Utilisation d'un bouton et d'un contrôle texte

du bouton et du contrôle texte, il faut supprimer l'appel à `shell.setLayout(new RowLayout( ))` et utiliser à la place la méthode `setBounds`, de la façon suivante :

```
public static void main (String [] args) {
   Display display = new Display ();
   Shell shell = new Shell (display);
   shell.setSize(300, 200);
   shell.setText("Button Exemple");

   final Button bouton = new Button(shell, SWT.PUSH);
   bouton.setText("Click Me");
   bouton.setBounds(80, 80, 90, 20);

   final Text texte = new Text (shell, SWT.SHADOW_IN);
   texte.setBounds(180, 80, 90, 20);
```

Ayant ajouté à l'application des éléments interactifs, il est indispensable d'en savoir un peu plus sur la façon dont ils sont gérés par les threads de l'application SWT. Dans une application SWT, le thread principal, le *UI thread* (interface utilisateur), est responsable de la gestion des événements qu'il redistribue aux bons éléments. Dans AWT et dans Swing, il n'est pas nécessaire d'agir sur ce thread du UI ; dans SWT, en revanche, ce thread agit comme un collecteur de messages qui renvoie les événements aux éléments qui en ont besoin (c'est ce mécanisme qui rend possible l'utilisation de modules SWT additionnels dans Eclipse). Le thread UI est la tâche principale de l'application et, lorsque la complexité augmente, il faut démarrer d'autres tâches pour tout faire. Cela étant, le UI thread est le seul qui peut interagir avec les éléments de l'interface graphique et ne pas émettre une exception `SWTException`.

Si plusieurs threads sont lancés, il est possible de mettre à jour l'interface utilisateur lorsque leur tâche est achevée, au moyen des méthodes `asyncExec` et `syncExec`. On transmet ces méthodes à une nouvelle instance d'un objet `Runnable` qui va réaliser la mise à jour voulue. Dans un thread de traitement, par exemple, il est possible d'atteindre l'objet `Display` courant, grâce à la méthode `Display.getDefault`, et d'attribuer un texte à une étiquette, en procédant comme suit :

```
Display.getDefault().asyncExec(new Runnable()
{
    public void run()
    {
        label.setText("Pas de souci !");
    }
});
```

Utilisation des éléments composites et des gestionnaires de placement

En sélectionnant un gestionnaire de placement, on a la possibilité de spécifier comment sont disposés les contrôles ; dans SWT, il existe quatre classes pour cette gestion :

`FillLayout`
: Permet de remplir complètement la zone de la fenêtre

`GridLayout`
: Dispose les contrôles selon une grille

`RowLayout`
: Dispose les contrôles selon des rangées horizontales ou bien des colonnes verticales

`FormLayout`
: Permet de positionner le contrôle relativement à un composite parent ou un autre contrôle. C'est le plus précis des gestionnaires de placement.

Par exemple, voici comment fonctionne le placement selon une grille. On crée une grille ayant quatre colonnes, puis on la remplit avec des boutons. Pour cet exemple, on crée un nouveau contrôle `Composite`, qui peut contenir d'autres contrôles et que l'on remplit avec les boutons. Pour commencer, on crée la fenêtre (shell) pour afficher le contrôle composite avec un placement en rangée. Le contrôle composite est ensuite créé (la constante `SWT.NONE` signifiant qu'aucun style particulier ne lui est affecté) ; à l'intérieur de celui-ci on installe le gestionnaire de placement selon une grille à quatre colonnes :

```
public static void main (String [] args) {
        Display display = new Display ();
        final Shell shell = new Shell (display);
        shell.setSize(300, 200);
        shell.setLayout(new RowLayout());

        final Composite composite = new Composite(shell, SWT.NONE);
        GridLayout gridLayout = new GridLayout();
        gridLayout.numColumns = 4;
        composite.setLayout(gridLayout);
    .
    .
    .
```

On complète par l'ajout des boutons dans le composite en utilisant une boucle et on termine avec la boucle de capture des événements, comme le montre l'exemple 7-3.

Exemple 7-3. Utilisation des gestionnaires de placement SWT, Ch07_03.java

```
package org.eclipsebook.ch07;

import org.eclipse.swt.*;
import org.eclipse.swt.layout.*;
import org.eclipse.swt.widgets.*;
```

Exemple 7-3. Utilisation des gestionnaires de placement SWT, Ch07_03.java (suite)

```
public class Ch07_03 {

        public static void main (String [] args) {
                Display display = new Display ();
                final Shell shell = new Shell (display);
                shell.setSize(300, 200);
                shell.setLayout(new GridLayout());

                final Composite composite = new Composite(shell, SWT.NONE);
                GridLayout gridLayout = new GridLayout();
                gridLayout.numColumns = 4;
                composite.setLayout(gridLayout);

                for (int loopIndex = 0; loopIndex < 18; loopIndex++) {
                        Button bouton = new Button(composite, SWT.PUSH);
                        bouton.setText("Bouton " + loopIndex);
                }

                shell.open ();
                while (!shell.isDisposed()) {
                        if (!display.readAndDispatch()) display.sleep();
                }
                display.dispose ();
        }
}
```

Le résultat est visible dans la figure 7-3, dans laquelle les boutons sont disposés selon une grille.

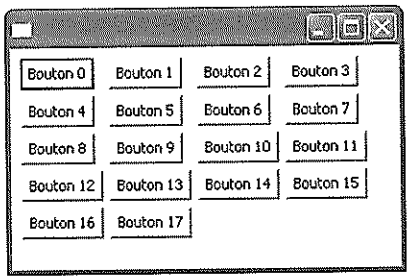


Figure 7-3. Utilisation du placement selon une grille

Le placement, relativement nouveau, de type *form layout* est très puissant car il permet le positionnement des contrôles par rapport à d'autres contrôles ou d'un conteneur.

Utilisation des Listes

La liste est un autre contrôle assez banal, représentée par la classe SWT `List`. Voici un exemple, utilisant ce contrôle, qui montre comment récupérer les sélections réalisées dans la liste. Une liste peut utiliser l'un des styles suivants :

`SWT.BORDER`
: Ajoute une bordure

`SWT.H_SCROLL`
: Ajoute une barre de défilement horizontale

`SWT.V_SCROLL`
: Ajoute une barre de défilement verticale

`SWT.SINGLE`
: Autorise la sélection unique

`SWT.MULTI`
: Autorise les sélections multiples

En utilisant *fill layout*, on remplit la totalité de la surface de la fenêtre avec le contrôle liste. Voici le code correspondant – noter l'usage de la méthode `add` de la classe `List` pour ajouter les éléments à la liste :

```
public class Ch07_04 {

        public static void main (String [] args) {
                Display display = new Display ();
                Shell shell = new Shell (display);
                shell.setText("Exemple de liste");
                shell.setSize(300, 200);
                shell.setLayout(new FillLayout(SWT.VERTICAL));

                final List liste = new List (shell, SWT.BORDER | SWT.MULTI |
                          SWT.V_SCROLL);

                for (int loopIndex = 0; loopIndex < 100; loopIndex++){
                    liste.add("Item " + loopIndex);
                }
        .
        .
        .
```

À présent, on peut récupérer les éléments sélectionnés en utilisant la capture de l'événement dû à la sélection et grâce à la méthode `getSelectionIndices`. Cette méthode retourne un tableau d'`int` contenant les indices des éléments sélectionnés dans le contrôle liste ; ces indices peuvent être affichés sur la console, comme le montre l'exemple 7-4.

Exemple 7-4. Utilisation des listes

```
package org.eclipsebook.ch07;
```

Exemple 7-4. Utilisation des listes (suite)

```
import org.eclipse.swt.*;
import org.eclipse.swt.widgets.*;
import org.eclipse.swt.layout.*;
import org.eclipse.swt.events.*;

public class Ch07_04 {

        public static void main (String [] args) {
             Display display = new Display ();
             Shell shell = new Shell (display);
             shell.setText("Exemple de liste");
             shell.setSize(300, 200);
             shell.setLayout(new FillLayout(SWT.VERTICAL));

             final List liste = new List (shell, SWT.BORDER | SWT.MULTI |
                         SWT.V_SCROLL);

             for (int loopIndex = 0; loopIndex < 100; loopIndex++){
                 liste.add("Item " + loopIndex);
                }

             list.addSelectionListener(new SelectionListener()
             {

                public void widgetSelected(SelectionEvent event)
                {
                     int [] selections = liste.getSelectionIndices ();
                     String outText = "";
                     for (int loopIndex = 0; loopIndex < selections.length;
                         loopIndex++) outText += selections[loopIndex] + " ";
                     System.out.println ("Vous avez sélectionné " + outText);
                   }

                public void widgetDefaultSelected(SelectionEvent event)
                {
                     int [] selections = liste.getSelectionIndices ();
                     String outText = "";
                     for (int loopIndex = 0; loopIndex < selections.length;
                           loopIndex++)
                         outText += selections[loopIndex] + " ";
                    System.out.println ("Vous avez sélectionné " + outText);
                  }
               });

               shell.open ();
               while (!shell.isDisposed ()) {
                   if (!display.readAndDispatch ()) display.sleep ();
               }
               display.dispose ();
        }
}
```

Exemple 7-4. Utilisation des listes (suite)

On peut voir, dans la figure 7-4, les différentes sélections de l'utilisateur réalisées dans le contrôle liste.

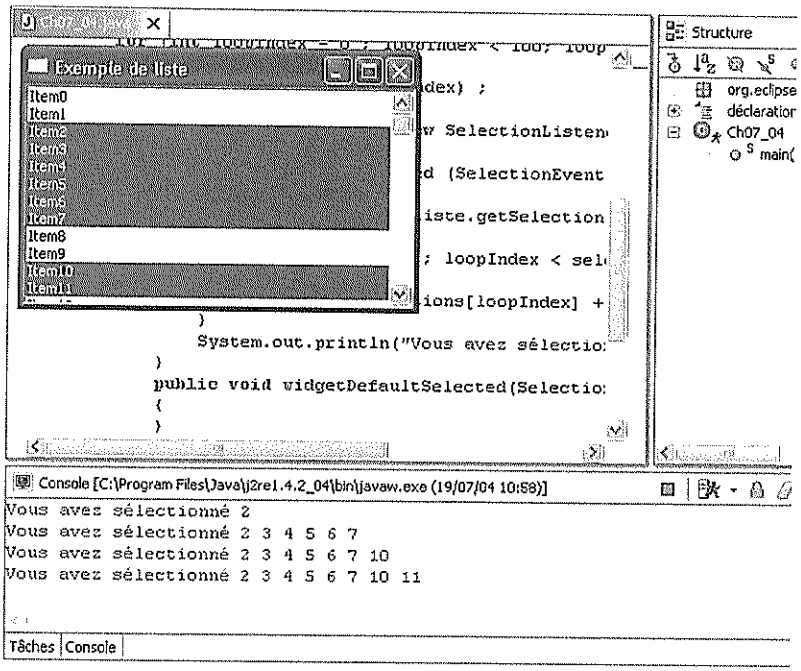


Figure 7-4. Sélections d'éléments dans une liste

Utilisation de V4ALL avec SWT

Il est possible d'utiliser V4ALL pour créer du code SWT, ce qui peut rendre les choses bien plus simples car il suffit de déposer les contrôles où l'on veut. La figure 7-5, illustre un exemple similaire à celui présenté pour Swing dans le précédent chapitre. Ici, on crée un nouvel éditeur V4ALL dans un projet nommé `Ch07_05`, exactement comme on l'a fait au chapitre 6 – mais cette fois on crée du code SWT et non pas du code Swing.

Après avoir créé le nouvel éditeur V4ALL, on ajoute à la nouvelle fenêtre un bouton et une étiquette, on change la légende du bouton avec Cliquez moi et l'on supprime la légende de l'étiquette. Ensuite, on dépose la méthode sur la feuille de dessin et on la nomme `processEvent`, comme on peut le voir sur la figure 7-5. Pour relier cette méthode au bouton, on utilise la vue Propriétés et, comme cela a déjà été fait au chapitre 6, on connecte la méthode `processEvent` à l'événement du bouton Action Performed.

Pour générer le code de l'application SWT, on commence par ajouter au chemin de compilation le fichier *swt.jar*, puis l'on sélectionne l'option de menu Code Generation → Generate Code for SWT, créant ainsi *Gui_1.java*. Dans ce fichier, Eclipse introduit

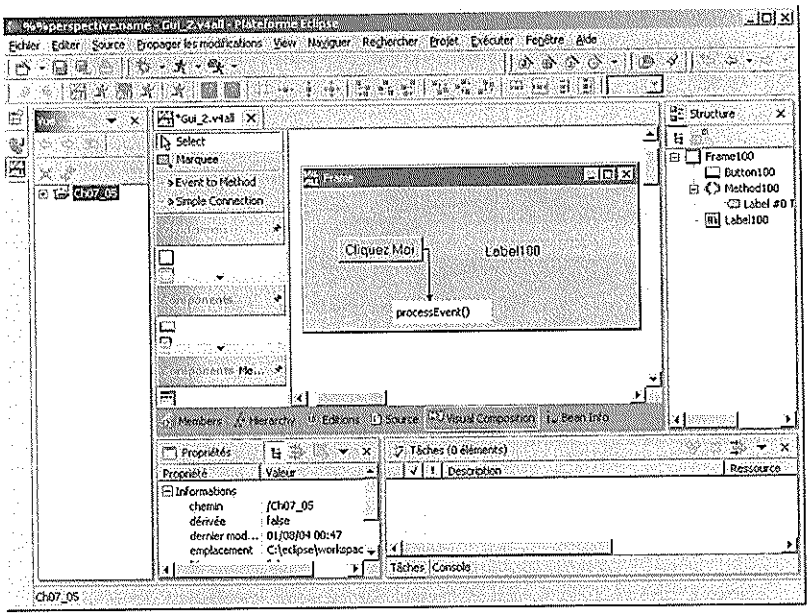


Figure 7-5. Conception d'une application SWT avec V4ALL

deux importations qui ne seront jamais utilisés, qui peuvent donc être supprimées :

```
import org.eclipse.swt.*;
import org.eclipse.swt.graphics.*;
import org.eclipse.swt.widgets.*;
import org.eclipse.swt.custom.*;
import org.eclipse.swt.layout.*;
```

On écrit ensuite le code de la méthode `processEvent`, qui inscrit le message « Pas de souci » dans l'étiquette lorsque l'utilisateur clique sur le bouton :

```
public void processEvent(){
// user code begin {1} Swing
// user code end
// user code begin {1} SWT
    ivjLabel100.setText("Pas de souci");
// user code end
// user code begin {1} HTML
// user code end
// user code begin {1} Eclipse
// user code end
// user code begin {1} C#
// user code end
}
```

Pour terminer, on crée une configuration d'exécution pour la classe `Gui_1` du fichier *Gui_1.java*, puis on donne les arguments de la VM avec `-Djava.library.path="d:\eclipse211\eclipse\plugins\org.eclipse.swt.win32_2.1.1\os\win32\x86"` comme cela a été vu précédemment dans ce chapitre. Quand

on lance cette application SWT, on obtient le résultat visible à la figure 7-6, dans lequel un clic sur le bouton provoque l'affichage du message dans l'étiquette.

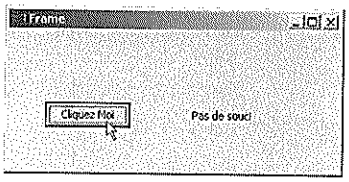


Figure 7-6. Exécution d'une application V4ALL SWT

Ainsi se termine ce premier chapitre sur SWT. Nous avons présenté les contrôles les plus importants, mais il en existe bien d'autres. Le chapitre suivant nous permettra d'en découvrir de nouveaux, notamment les menus et les sélecteurs.

CHAPITRE 8

SWT : menus, barres d'outils, curseurs, arbres et boîtes de dialogue

SWT dispose de nombreux éléments graphiques, au-delà de ceux qui ont été abordés au chapitre précédent ; le présent chapitre propose de découvrir en détail les contrôles les plus importants de SWT : menus, barre d'outils, curseurs, arbres, et dialogues personalisés.

Utilisation des menus

Comme tout générateur d'interface graphique, SWT gère bien entendu les menus. Le processus de création et de gestion des menus d'une application SWT n'est pas forcément simple mais pas excessivement difficile. Nous allons nous familiariser avec ce type de travail grâce à un exemple dans lequel on crée un système de menus ayant un menu Fichier et un menu d'Aide, et affichant dans un champ texte le nom de l'option sélectionnée.

Dans SWT, pour créer un système de menus, on crée un objet `Menu` correspondant au sommet de la hiérarchie du système de menus. Comme pour tous les contrôles de SWT, on peut attribuer divers styles aux menus ; le tableau 7-1 en fait l'inventaire :

`SWT.BAR`
: Attribue le comportement d'une barre de menu

`SWT.DROP_DOWN`
: Crée un menu déroulant

`SWT.POP_UP`
: Crée un menu surgissant

`SWT.NO_RADIO_GROUP`
: Empêche le regroupement de boutons radio

`SWT.LEFT_TO_RIGHT`
: Attribue l'orientation de gauche à droite

`SWT.RIGHT_TO_LEFT`
: Attribue l'orientation de droite à gauche

Les menus Fichier et d'Aide sont des objets MenuItem; voici les styles possibles pour les options de menu:

SWT.CHECK
: Crée une case à cocher

SWT.CASCADE
: Crée un sous-menu en cascade

SWT.PUSH
: Crée un bouton poussoir

SWT.RADIO
: Crée un bouton radio

SWT.SEPARATOR
: Crée un séparateur de menu

On notera, en particulier, la simplicité pour ajouter un séparateur dans un menu, uniquement en créant une option de menu ayant le style SWT.SEPARATOR.

Pour créer le menu Fichier, on écrit son nom en faisant précéder – comme dans d'autres paquetages pour interface graphique, la lettre F de Fichier par le signe &, ce qui définit un raccourci (permettant l'ouverture du menu, en pressant Alt+F sous Windows ou Pomme+F sous Mac Os X) :

```
menuBar = new Menu(shell, SWT.BAR);
fileMenuHeader = new MenuItem(menuBar, SWT.CASCADE);
fileMenuHeader.setText("&Fichier");
```

Cela ajoute l'option Fichier à la barre de menu. La prochaine étape permet de créer le menu déroulant qui affiche toutes les options du menu Fichier ; on l'écrit ainsi :

```
menuBar = new Menu(shell, SWT.BAR);
fileMenuHeader = new MenuItem(menuBar, SWT.CASCADE);
fileMenuHeader.setText("&Fichier");

fileMenu = new Menu(shell, SWT.DROP_DOWN);
fileMenuHeader.setMenu(fileMenu);
```

Dans cet exemple, on attribue deux items au menu Fichier, Enregistrer et Quitter. Il y a donc deux objets MenuItem à créer comme ceci :

```
menuBar = new Menu(shell, SWT.BAR);
fileMenuHeader = new MenuItem(menuBar, SWT.CASCADE);
fileMenuHeader.setText("&Fichier");

fileMenu = new Menu(shell, SWT.DROP_DOWN);
fileMenuHeader.setMenu(fileMenu);

fileSaveItem = new MenuItem(fileMenu, SWT.PUSH);
fileSaveItem.setText("&Enregistrer");

fileExitItem = new MenuItem(fileMenu, SWT.PUSH);
fileExitItem.setText("&Quitter");
```

Les options de menu Fichier → Enregistrer et Fichier → Quitter sont gérés par des récepteurs d'événements. Voici comment relier ces options aux objets du type SelectionListener, fileSaveItemListener et fileExitItemListener :

```
fileSaveItem.addSelectionListener(new fileSaveItemListener());
fileExitItem.addSelectionListener(new fileExitItemListener());
```

Quand l'option Fichier → Enregistrer est sélectionnée, on veut que le message « Demande de sauvegarde » s'affiche dans un champ texte :

```
class fileSaveItemListener implements SelectionListener
{
   public void widgetSelected(SelectionEvent event)
   {
    label.setText("Demande de sauvegarde");
   }

   public void widgetDefaultSelected(SelectionEvent event)
   {
    label.setText("Demande de sauvegarde");
   }
}
```

Par ailleurs, quand l'utilisateur sélectionne l'option de menu Fichier → Quitter, nous voulons quitter l'application. Informatiquement parlant, la fenêtre (shell) est fermée et l'objet display est libéré :

```
class fileExitItemListener implements SelectionListener
{
   public void widgetSelected(SelectionEvent event)
   {
      shell.close();
      display.dispose();
   }

   public void widgetDefaultSelected(SelectionEvent event)
   {
      shell.close();
      display.dispose();
   }
}
```

Quand la préparation du système de menus est terminée, il faut ajouter celui-ci à l'objet menuBar de la fenêtre courante, au moyen de la méthode setMenuBar, puis déclencher l'affichage de la fenêtre comme d'habitude :

```
shell.setMenuBar(menuBar);

shell.open();
while(!shell.isDisposed()) {
    if(!display.readAndDispatch()) display.sleep();
}
display.dispose();
```

Voilà pour le menu Fichier. Nous voulons maintenant ajouter l'option de menu Aide → Au secours ; le code supplémentaire est donné dans le listing de l'exemple 8-1.

Exemple 8-1. Utilisation des menus SWT

```
package org.eclipsebook.ch08;

import org.eclipse.swt.widgets.*;
import org.eclipse.swt.SWT;
import org.eclipse.swt.events.*;

public class Ch08_01 {

    Display display;
    Shell shell;
    Menu menuBar, fileMenu, helpMenu;
    MenuItem fileMenuHeader, helpMenuHeader;
    MenuItem fileExitItem, fileSaveItem, helpGetHelpItem;
    Label label;

    public Ch08_01(){

        display = new Display();
        shell = new Shell(display);
        shell.setText("Exemple de menu");
        shell.setSize(300, 200);

        label = new Label(shell, SWT.CENTER);
        label.setBounds(shell.getClientArea());

        menuBar = new Menu(shell, SWT.BAR);
        fileMenuHeader = new MenuItem(menuBar, SWT.CASCADE);
        fileMenuHeader.setText("&Fichier");

        fileMenu = new Menu(shell, SWT.DROP_DOWN);
        fileMenuHeader.setMenu(fileMenu);

        fileSaveItem = new MenuItem(fileMenu, SWT.PUSH);
        fileSaveItem.setText("&Enregistrer");

        fileExitItem = new MenuItem(fileMenu, SWT.PUSH);
        fileExitItem.setText("&Quitter");

        helpMenuHeader = new MenuItem(menuBar, SWT.CASCADE);
        helpMenuHeader.setText("&Aide");

        helpMenu = new Menu(shell, SWT.DROP_DOWN);
        helpMenuHeader.setMenu(helpMenu);

        helpGetHelpItem = new MenuItem(helpMenu, SWT.PUSH);
        helpGetHelpItem.setText("Au &secours");

        fileExitItem.addSelectionListener(new fileExitItemListener());
        fileSaveItem.addSelectionListener(new fileSaveItemListener());
        helpGetHelpItem.addSelectionListener(new helpGetHelpItemListener());
```

Exemple 8-1. Utilisation des menus SWT (suite)

```
        shell.setMenuBar(menuBar);
        shell.open();
        while(!shell.isDisposed()) {
            if(!display.readAndDispatch()) display.sleep();
        }
        display.dispose();
    }

    class fileExitItemListener implements SelectionListener
    {
       public void widgetSelected(SelectionEvent event)
       {
          shell.close();
          display.dispose();
       }

       public void widgetDefaultSelected(SelectionEvent event)
       {
          shell.close();
          display.dispose();
       }
    }

    class fileSaveItemListener implements SelectionListener
    {
       public void widgetSelected(SelectionEvent event)
       {
        label.setText("Demande de sauvegarde");
       }

       public void widgetDefaultSelected(SelectionEvent event)
       {
        label.setText("Demande de sauvegarde");
       }
    }

    class helpGetHelpItemListener implements SelectionListener
    {
       public void widgetSelected(SelectionEvent event)
       {
          label.setText("Appel au secours");
       }

       public void widgetDefaultSelected(SelectionEvent event)
       {
        label.setText("Appel au secours");
       }
    }

    public static void main(String [] args) {
```

Exemple 8-1. Utilisation des menus SWT (suite)

```
            Ch08_01 menuExample = new Ch08_01();
        }
    }
```

Le résultat est visible dans la figure 8-1 et la figure 8-2 ; quand l'utilisateur sélectionne l'option de menu Fichier → Enregistrer, l'application récupère cette sélection et affiche le message dans le champ texte qui a été ajouté à la fenêtre.

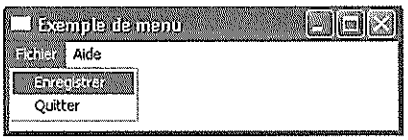


Figure 8-1. Sélection d'une option de menu SWT

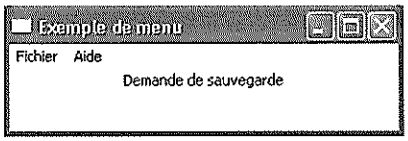


Figure 8-2. Résultat de la sélection de l'option de menu

Une des options les plus classiques du système de menus concerne la boîte de dialogue A propos de l'application, que l'on sélectionne avec Aide → A propos. Pour que l'application dispose de l'option de menu Aide → A propos, le lecteur se reportera à la mise en œuvre des boîtes de dialogue plus loin dans ce chapitre. Quand cette option est sélectionnée, il faut ouvrir une boîte de dialogue qui contient tous les contrôles nécessaires à l'affichage des informations de l'application.

Utilisation de la barre d'outils

On peut mettre en œuvre les barres d'outils dans SWT en utilisant la classe `Toolbar`. Pour créer une barre d'outils, il suffit de l'associer à la fenêtre avec laquelle on travaille, puis d'en définir les styles :

`SWT.BORDER`
: Barre d'outils avec une bordure

`SWT.FLAT`
: Barre d'outils plate

`SWT.WRAP`
: Barre d'outils qui peut se redessiner sur plusieurs lignes

`SWT.RIGHT`
: Barre d'outils alignée sur la droite

`SWT.SHADOW_OUT`
Barre d'outils décorée d'une ombre

`SWT.HORIZONTAL`
Barre d'outils horizontale

`SWT.VERTICAL`
Barre d'outils verticale

Voici comment créer une nouvelle barre d'outils (la constante `SWT.NONE` signifie qu'aucun style particulier n'est choisi) :

```
ToolBar toolbar = new ToolBar(shell, SWT.NONE);
```

Chaque item de la barre d'outils est un objet `ToolItem` ; dans cet exemple, on crée quatre boutons poussoirs en utilisant le style `SWT.PUSH` et la classe `ToolItem`. Voici les styles possibles pour les items d'outil :

`SWT.PUSH`
Un bouton poussoir

`SWT.CHECK`
Une case à cocher

`SWT.RADIO`
Un bouton radio

`SWT.SEPARATOR`
Un séparateur de barre d'outils

`SWT.DROP_DOWN`
Un élément déroulant

On peut fixer le texte apparaissant dans chaque bouton – dans ce cas, on inscrit successivement `A présent`, `il`, `est`, `temps` – comme ceci :

```
ToolBar toolbar = new ToolBar(shell, SWT.NONE);
ToolItem item1 = new ToolItem(toolbar, SWT.PUSH);
item1.setText("A présent");
ToolItem item2 = new ToolItem(toolbar, SWT.PUSH);
item2.setText("il");
ToolItem item3 = new ToolItem(toolbar, SWT.PUSH);
item3.setText("est");
ToolItem item4 = new ToolItem(toolbar, SWT.PUSH);
item4.setText("temps");
```

On peut aussi afficher dans la barre d'outils des images, des images inactives et des images actives (lesquelles s'affichent lorsque la souris passe sur l'item de l'outil) en utilisant un objet `Image` de SWT :

```
ToolItem item1 = new ToolItem(toolbar, SWT.PUSH);
item1.setImage(image);
item1.setDisabledImage(disabledImage);
item1.setHotImage(hotImage);
```

Dans la fenêtre, on définit les caractéristiques géométriques de la barre d'outils et du champ texte que l'on crée pour afficher la légende du bouton que l'utilisateur a pressé :

```
toolbar.setBounds(0, 0, 200, 70);

final Text text = new Text(shell, SWT.BORDER);
text.setBounds(0, 100, 200, 25);
```

Il est nécessaire de relier les items de la barre d'outils aux récepteurs d'événements qui vont gérer leurs événements ; ici, on utilise un objet générique `Listener`. On va pouvoir gérer les événements provenant des items de la barre d'outils en utilisant la méthode `handleEvent` de la classe `Listener` :

```
Listener listener = new Listener() {
    public void handleEvent(Event event) {
        .
        .
        .
    }
};
```

Cette méthode reçoit un objet `event` à partir duquel on extrait, grâce à la donnée membre `widget`, l'élément de l'interface qui est la cause de l'événement. Ayant identifié l'item de la barre d'outils, on peut en extraire le texte associé grâce à la méthode `getText`, qui renvoie le texte de n'importe quel élément d'interface, du contrôle texte jusqu'au bouton de barre d'outils :

```
Listener listener = new Listener() {
    public void handleEvent(Event event) {
        ToolItem item = (ToolItem)event.widget;
        String string = item.getText();
            .
            .
            .
    }
};
```

Pour terminer ce gestionnaire d'événements, l'item de la barre d'outils étant identifié, on affiche le message idoine dans le champ texte – vous noterez que l'on récupère la légende du bouton, car on sait lequel a été cliqué :

```
Listener listener = new Listener() {
    public void handleEvent(Event event) {
        ToolItem item =(ToolItem)event.widget;
        String string = item.getText();
        text.setText("Vous avez sélectionné : " + string);
    }
};
```

Ainsi se termine le gestionnaire d'événements que l'on peut attacher à chaque item d'outil avec la méthode `addListener`. Nous allons maintenant voir comment préciser que l'on veut gérer l'événement `SWT.Selection`, comme le montre le listing de l'exemple 8-2.

Exemple 8-2. Utilisation d'une barre d'outils SWT

```
package org.eclipse.ch08;

import org.eclipse.swt.*;
import org.eclipse.swt.widgets.*;

public class Ch08_02 {

        public static void main(String [] args) {
            Display cisplay = new Display();
            final Shell shell = new Shell(display);
            shell.setSize(300, 200);

            ToolBar toolbar = new ToolBar(shell, SWT.NONE);
            ToolItem item1 = new ToolItem(toolbar, SWT.PUSH);
            item1.setText("A présent");
            ToolItem item2 = new ToolItem(toolbar, SWT.PUSH);
            item2.setText("il");
            ToolItem item3 = new ToolItem(toolbar, SWT.PUSH);
            item3.setText("est");
            ToolItem item4 = new ToolItem(toolbar, SWT.PUSH);
            item4.setText("temps");

             toolbar.setBounds(0, 0, 200, 70);

            final Text text = new Text(shell, SWT.BORDER);
            text.setBounds(0, 100, 200, 25);

            Listener listener = new Listener() {
                public void handleEvent(Event event) {
                    ToolItem item =(ToolItem)event.widget;
                    String string = item.getText();
                    text.setText("Vous avez sélectionné : " +
                           string);              }
            };

            item1.addListener(SWT.Selection, listener);
            item2.addListener(SWT.Selection, listener);
            item3.addListener(SWT.Selection, listener);
            item4.addListener(SWT.Selection, listener);

            shell.open();

            while (!shell.isDisposed()) {
                if (!display.readAndDispatch())
                    display.sleep();
            }
            display.dispose();
        }
}
```

Le résultat apparaît dans la figure 8-3. La barre d'outils est un élément d'interface comme un autre ; ici, elle est placée en haut à gauche de la fenêtre. On peut la

positionner grâce à la méthode setBounds ou encore grâce à l'utilisation d'un gestionnaire de placement de SWT.

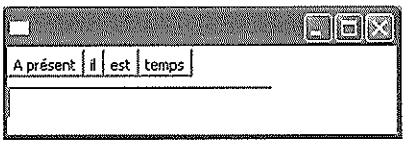


Figure 8-3. Une nouvelle barre d'outils

Comme on peut le voir sur la figure 8-4, quand l'utilisateur clique sur l'un des outils, l'application en rend compte correctement.

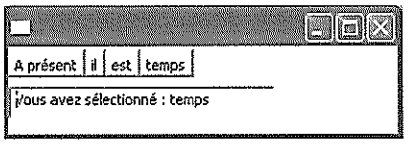


Figure 8-4. Reconnaissance d'un événement provenant de la barre d'outils

Il convient de noter que l'on peut élaborer des barre d'outils dans lesquelles on installe n'importe quel contrôle – par exemple, pour qu'une barre d'outils contienne un sélecteur d'option (combo box), on peut utiliser le code suivant :

```
Combo combo = new Combo(toolbar, SWT.READ_ONLY);
```

Comme tout bon paquetage graphique qui se respecte, la barre d'outils SWT peut aussi afficher des menus déroulants. Dans SWT, on utilise cependant des menus de barre d'outils avec des menus contextuel, dont on fixe la position et la visibilité en utilisant nécessairement un code comme celui-ci :

```
menu.setLocation(point.x, point.y);
menu.setVisible(true);
```

Utilisation de curseurs

Le curseur est un autre contrôle SWT bien pratique et très facile à utiliser, qui permet à l'utilisateur de choisir une valeur numérique dans un intervalle. Dans l'exemple suivant, on relève sa nouvelle position lorsque l'utilisateur en déplace l'index (aussi appelé curseur de boîte de défilement). Ici, les styles possibles, lorsque l'on crée un curseur, sont :

`SWT.BORDER`
: Ajoute une bordure

`SWT.HORIZONTAL`
: Crée un curseur horizontal

`SWT.VERTICAL`
: Crée un curseur vertical

Pour inviter l'utilisateur à interagir, on place une étiquette (« Déplacer le curseur »), puis on dépose un curseur utilisant le style SWT.HORIZONTAL (l'utilisation de SWT.VERTICAL créant bien entendu un curseur vertical) ; enfin, on place un champ texte afin d'afficher la nouvelle position du curseur :

```
final Label label = new Label(shell, SWT.NONE);
label.setText("Déplacer le curseur");
label.setBounds(0, 20, 150, 15);

final Slider slider = new Slider(shell, SWT.HORIZONTAL);
slider.setBounds(0, 40, 200, 20);

final Text text = new Text(shell, SWT.BORDER);
text.setBounds(0, 100, 200, 25);
```

Pour finaliser la mise en place du curseur, il faut ensuite gérer les actions de l'utilisateur. Les curseurs acceptent un certain nombre d'événements, chacun d'eux étant codé par une constante SWT :

SWT.ARROW_DOWN
: Le bouton en forme de flèche bas/droite a été cliqué.

SWT.ARROW_UP
: Le bouton en forme de flèche haut/gauche a été cliqué.

SWT.DRAG
: L'index du curseur a été déplacé.

SWT.END
: Le curseur est arrivé à la fin de sa course.

SWT.HOME
: Le curseur est arrivé au début de sa course.

SWT.PAGE_DOWN
: La barre de défilement bas/droite a été cliquée.

SWT.PAGE_UP
: La barre de défilement haut/gauche a été cliquée.

On utilise le membre detail d'un objet event pour déterminer lequel de ces événements s'est produit ; dans le le code suivant, on utilise une forme switch pour afficher dans le champ texte l'événement qui a été reçu :

```
slider.addListener(SWT.Selection, new Listener() {
        public void handleEvent(Event event) {
                String outString = "Event: SWT.NONE";
                switch(event.detail) {
                case SWT.ARROW_DOWN: outString = "Evénement: SWT.ARROW_DOWN";
                break;
                case SWT.ARROW_UP: outString = "Evénement: SWT.ARROW_UP";
                break;
                case SWT.DRAG: outString = "Evénement: SWT.DRAG";
                break;
                case SWT.END: outString = "Evénement: SWT.END";
```

```
                break;
                case SWT.HOME: outString = "Evénement: SWT.HOME";
                break;
                case SWT.PAGE_DOWN: outString = "Evénement: SWT.PAGE_DOWN";
                break;
                case SWT.PAGE_UP: outString = "Evénement: SWT.PAGE_UP";
                break;
                                }
            .
            .
            .
        }
});
```

Pour déterminer la position courante du curseur, on utilise la méthode getSelection, comme le montre le listing de l'exemple 8-3.

Exemple 8-3. Utilisation des curseurs SWT

```
package org.eclipsebook.ch08;

import org.eclipse.swt.*;
import org.eclipse.swt.widgets.*;

public class Ch08_03 {

   public static void main(String [] args) {
        Display display = new Display();
        Shell shell = new Shell(display);
        shell.setText("Curseurs");
        shell.setSize(300, 200);

        final Label label = new Label(shell, SWT.NONE);
        label.setText("Déplacer le curseur");
        label.setBounds(0, 20, 180, 15);

        final Slider slider = new Slider(shell, SWT.HORIZONTAL);
        slider.setBounds(0, 40, 250, 20);

        final Text text = new Text(shell, SWT.BORDER);
        text.setBounds(0, 100, 250, 25);

        slider.addListener(SWT.Selection, new Listener() {
            public void handleEvent(Event event) {
               String outString = "Evénement: SWT.NONE";
               switch(event.detail) {
                   case SWT.ARROW_DOWN: outString = "Evénement: SWT.ARROW_DOWN";
                   break;
                   case SWT.ARROW_UP: outString = "Evénement: SWT.ARROW_UP";
                   break;
                   case SWT.DRAG: outString = "Evénement: SWT.DRAG";
                   break;
                   case SWT.END: outString = "Evénement: SWT.END";
                   break;
```

Exemple 8-3. Utilisation des curseurs SWT (suite)

```
                    case SWT.HOME: outString = "Evénement: SWT.HOME";
                    break;
                    case SWT.PAGE_DOWN: outString = "Evénement: SWT.PAGE_DOWN";
                    break;
                    case SWT.PAGE_UP: outString = "Evénement: SWT.PAGE_UP";
                    break;
                }
                outString += " Position: " + slider.getSelection();
                text.setText(outString);
            }
        });

        shell.open();
        while(!shell.isDisposed()) {
            if(!display.readAndDispatch()) display.sleep();
        }
        display.dispose();
    }
}
```

On peut observer le résultat obtenu dans la figure 8-5 : pendant que l'utilisateur déplace l'index du curseur, l'indication du code de l'événement et la nouvelle position de l'index sont affichées.

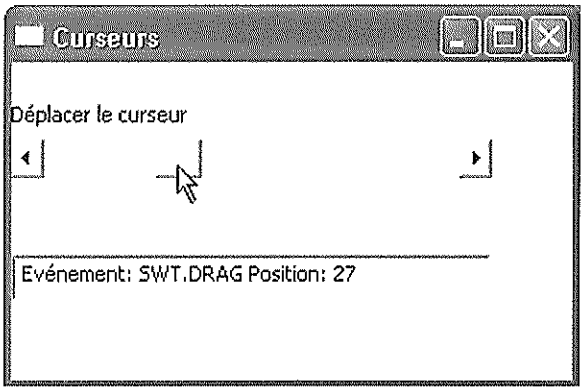


Figure 8-5. Déplacement de l'index du curseur

Quand l'utilisateur clique sur la barre de défilement, c'est un événement page up ou page down qui apparaît, ce que montre la figure 8-6 : l'index se déplace sur une nouvelle position (l'incrément de page est de 10 par défaut).

Outre l'utilisation de `getSelection` faite ci-dessus, on peut utiliser la méthode `setSelection` pour fixer la position courante du curseur. Par exemple, si l'utilisateur sélectionne une page vers laquelle il veut se déplacer dans un document, on peut modifier la position du curseur au moyen de la méthode `setSelection`.

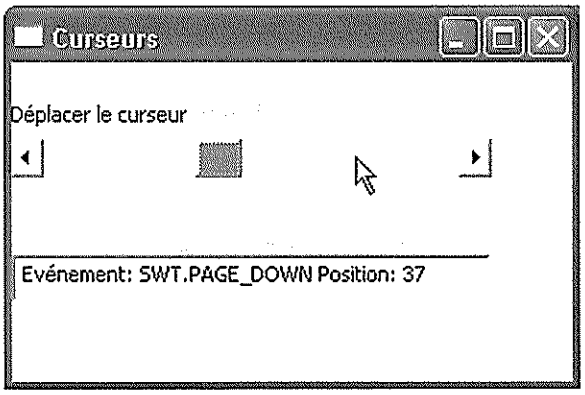


Figure 8-6. Création d'un événement page down

Par défaut, l'intervalle de valeurs du curseur s'étend de 0 à 100 moins la taille de l'index (qui est de 10 pixels par défaut). On peut fixer les valeurs minimum et maximum de l'intervalle du curseur grâce aux méthodes `setMinimum` et `setMaximum`. La valeur de l'incrément, lorsque les boutons en forme de flèche sont cliqués, peut être modifiée grâce à la méthode `setIncrement` ; l'incrément de page, lorsque les barres de défilement sont cliquées, peut l'être avec la méthode `setPageIncrement`. On peut aussi fixer la position du curseur avec la méthode `setPosition`.

Utilisation des arbres

Les arbres sont d'autres éléments standard de SWT ; on utilise la classe `Tree` pour créer des arbres et la classe `TreeItem` pour créer les nœuds d'un arbre. Voici les styles possibles pour les arbres :

`SWT.BORDER`
: Ajoute une bordure

`SWT.H_SCROLL`
: Ajoute une barre de défilement horizontale

`SWT.V_SCROLL`
: Ajoute une barre de défilement verticale

`SWT.SINGLE`
: Accepte la sélection unique

`SWT.MULTI`
: Accepte les sélections multiples

`SWT.CHECK`
: Accepte les cases à cocher

Dans cet exemple, on crée un nouvel arbre avec une bordure :

```
final Tree tree = new Tree(shell, SWT.BORDER);
tree.setSize(290, 290);
shell.setSize(300, 300);
```

Pour ajouter des nœuds à l'arbre, on utilise la classe `TreeItem` en passant l'objet `tree` au constructeur de la classe `TreeItem`. Dans cet exemple, on ajoute cinq nœuds au sommet de l'arbre :

```
final Tree tree = new Tree(shell, SWT.BORDER);
tree.setSize(290, 290);
shell.setSize(300, 300);

for(int loopIndex1 = 0; loopIndex1 < 5; loopIndex1++) {
    TreeItem item0 = new TreeItem(tree, 0);
    item0.setText("Level 0 Item " + loopIndex1);
    .
    .
    .
}
```

Chaque nœud de l'arbre est lui-même le point d'entrée d'une arborescence ; et les nœuds inférieurs en sont éventuellement d'autres, etc. Pour créer une arborescence sur un nœud, on transmet l'objet parent au constructeur de `TreeItem` du nœud enfant. Par exemple, on ajoute ici à chaque nœud, cinq nœuds enfants, lesquels à leur tour en possèdent cinq aussi :

```
for(int loopIndex1 = 0; loopIndex1 < 5; loopIndex1++) {
    TreeItem item0 = new TreeItem(tree, 0);
    item0.setText("Niveau 0 Item " + loopIndex1);
    for(int loopIndex2 = 0; loopIndex2 < 5; loopIndex2++) {
        TreeItem item1 = new TreeItem(item0, 0);
        item1.setText("Niveau 1 Item " + loopIndex2);
        for(int loopIndex3 = 0; loopIndex3 < 5; loopIndex3++) {
            TreeItem item2 = new TreeItem(item1, 0);
            item2.setText("Niveau 2 Item " + loopIndex3);
        }
    }
}
```

Voilà qui crée un arbre ; tout ce qui reste à faire c'est d'afficher la fenêtre. On peut lire la totalité du listing dans l'exemple 8-4.

Exemple 8-4. Utilisation d'arbres SWT

```
package org.eclipsebook.ch08;

import org.eclipse.swt.*;
import org.eclipse.swt.widgets.*;

public class Ch08_04 {

    public static void main(String [] args) {
        Display display = new Display();
        Shell shell = new Shell(display);
        shell.setText("Arbres");
```

Exemple 8-4. Utilisation d'arbres SWT (suite)

```
            final Tree tree = new Tree(shell, SWT.BORDER);
            tree.setSize(290, 290);
            shell.setSize(300, 300);

            for(int loopIndex1 = 0; loopIndex1 < 5; loopIndex1++) {
                TreeItem item0 = new TreeItem(tree, 0);
                item0.setText("Niveau 0 Item " + loopIndex1);
                for(int loopIndex2 = 0; loopIndex2 < 5; loopIndex2++) {
                    TreeItem item1 = new TreeItem(item0, 0);
                    item1.setText("Niveau 1 Item " + loopIndex2);
                    for(int loopIndex3 = 0; loopIndex3 < 5; loopIndex3++) {
                            TreeItem item2 = new TreeItem(item1, 0);
                            item2.setText("Niveau 2 Item " + loopIndex3);
                    }
                }
            }

            shell.open();
            while(!shell.isDisposed()) {
                if(!display.readAndDispatch()) display.sleep();
            }
            display.dispose();
        }
    }
```

Lorsque la première fenêtre s'ouvre, on peut voir cinq nœuds au sommet de l'arbre, comme le montre la figure 8-7.

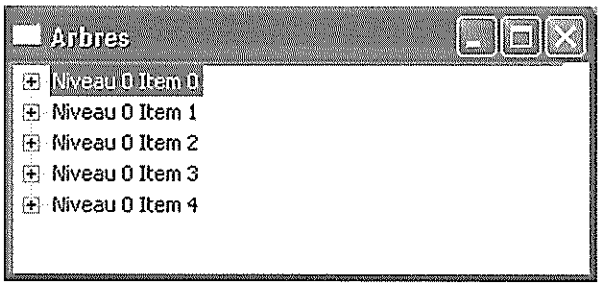


Figure 8-7. Un arbre clos

En cliquant sur une case +, on ouvre le nœud correspondant, comme l'illustre la figure 8-8.

Comme on peut le voir, il est assez aisé d'ajouter des nœuds à un arbre. Les arbres sont des contrôles élaborés – on peut y adjoindre des cases à cocher et utiliser les méthodes `setChecked` et `getChecked`, tout comme des images avec la méthode `setImage`. On peut aussi gérer les événements de sélection dans un arbre – l'objet `TreeItem` est capable de gérer les événements de sélection, exactement comme le font les objets `ToolbarItem` vus précédemment; il faut juste penser à lui ajouter le récepteur d'événements (listener).

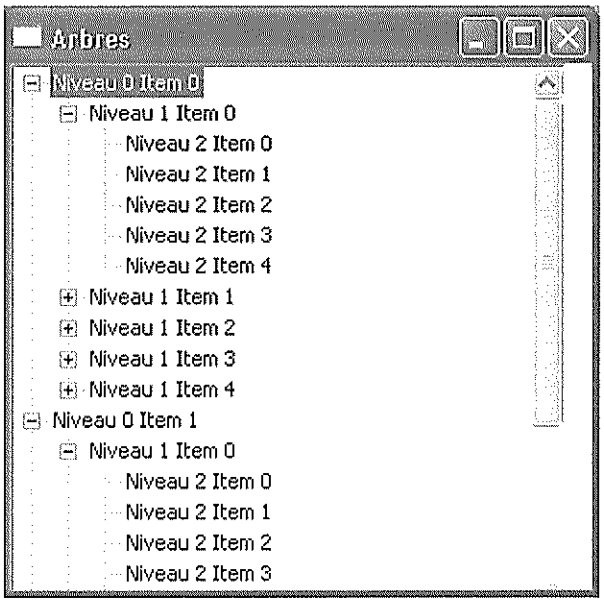


Figure 8-8. Un arbre ouvert

Utilisation des boîtes de dialogue

Comme dernier exemple SWT, nous allons créer de nouvelles boîtes de dialogue SWT et récupérer les données que l'utilisateur va y entrer. Il existe déjà dans SWT un certain nombre de classes prêtes à l'emploi, telles que `FileDialog` et `DirectoryDialog`, mais l'objectif de cet exemple est de concevoir des boîtes de dialogue personnalisées. La première affiche le message « OK pour avancer? », jusqu'à ce que l'on clique sur l'un des boutons OK ou Annuler, et détermine sur lequel des boutons l'utilisateur a cliqué.

Dans la fenêtre principale de l'application, on affiche un bouton nommé opener, sur lequel l'utilisateur doit cliquer pour faire apparaître une boîte de dialogue et un champ texte pour y afficher la sélection réalisée par l'utilisateur :

```
final Button opener = new Button(shell, SWT.PUSH);
opener.setText("Cliquez Moi");
opener.setBounds(20, 20, 50, 25);

final Text text = new Text(shell, SWT.SHADOW_IN);
text.setBounds(80, 20, 100, 25);
```

La boîte de dialogue est en soi un objet de type `Shell`. Voici les styles que l'on peut utiliser :

`SWT.APPLICATION_MODAL`
: Construit le dialogue comme une fenêtre d'application auxiliaire (boîte de dialogue modale)

`SWT.BORDER`
: Ajoute une bordure

`SWT.H_SCROLL`
Ajoute une barre de défilement horizontale

`SWT.V_SCROLL`
Ajoute une barre de défilement verticale

`SWT.CLOSE`
Ajoute le bouton de fermeture

`SWT.DIALOG_TRIM`
Style de la boîte de dialogue

`SWT.MIN`
Ajoute le bouton de minimisation

`SWT.MAX`
Ajoute le bouton de maximalisation

`SWT.NO_TRIM`
Interdit la mise en place de boutons supplémentaires

`SWT.RESIZE`
Gère le mouvement et le redimensionnement

`SWT.TITLE`
Prévoit la place pour le titre

Pour créer cette boîte de dialogue, on utilise les styles `SWT.APPLICATION_MODAL | SWT.DIALOG_TRIM`. Le choix du style « application modal » signifie que l'utilisateur doit fermer cette boîte de dialogue avant de pouvoir retravailler avec toute autre fenêtre de l'application :

```
final Shell dialog = new Shell(shell, SWT.APPLICATION_MODAL |
    SWT.DIALOG_TRIM);
dialog.setText("Dialogues");
dialog.setSize(150, 100);
```

Dans cet exemple, la boîte de dialogue comporte une étiquette (« OK pour avancer? ») invitant l'utilisateur à interagir avec les deux boutons OK et Annuler. On les ajoute alors au moyen du code suivant :

```
final Label label = new Label(dialog, SWT.NONE);
label.setText("OK pour avancer?");
label.setBounds(35, 5, 100, 20);

final Button okButton = new Button(dialog, SWT.PUSH);
okButton.setBounds(20, 35, 40, 25);
okButton.setText("OK");

Button cancelButton = new Button(dialog, SWT.PUSH);
cancelButton.setBounds(70, 35, 60, 25);
cancelButton.setText("Annuler");
```

Pour gérer les boutons OK et Annuler, on utilise un objet `Listener`. Ici, la difficulté réside dans l'utilisation de classes internes car il y est impossible de modifier une

variable de la classe englobante. En revanche, il est *possible* d'écrire, dans la classe interne, un code qui place des données dans un tableau défini dans la classe englobante ; c'est ce que l'on fait ici. Voici comment on détermine le bouton qui a été enfoncé et comment on positionne le booléen `response[0]` de façon cohérente avec le code principal de l'application – on notera que la boîte de dialogue est aussi fermée, suite à l'action sur les boutons OK ou Annuler :

```
final boolean [] response = new boolean[1];
response[0] = true;

Listener listener = new Listener() {
     public void handleEvent(Event event) {
            if(event.widget == okButton){
               response[0] = true;
            }else{
               response[0] = false;
            }
            dialog.close();
       }
};
```

À présent on peut attacher le récepteur d'événements aux boutons OK et Annuler. On attache aussi un récepteur au bouton opener qui ouvre et affiche la boîte de dialogue au premier plan :

```
okButton.addListener(SWT.Selection, listener);
cancelButton.addListener(SWT.Selection, listener);

Listener openerListener = new Listener() {
    public void handleEvent(Event event) {
        dialog.open();
        }
};

opener.addListener(SWT.Selection, openerListener);
```

Bien entendu, il est aussi nécessaire de prévoir la boucle de capture des événements de la boîte de dialogue. Enfin, lorsque celle-ci est fermée, on peut tester la valeur de `response[0]` et afficher le message précisant le bouton cliqué :

```
while(!dialog.isDisposed()) {
        if(!display.readAndDispatch()) display.sleep();
}

if(response[0]){
        text.setText("Vous avez cliqué OK");
} else {
        text.setText("Vous avez cliqué Annuler");
}
```

Ainsi se termine le code, dont on peut voir le listing complet dans l'exemple 8-5.

Exemple 8-5. Création d'un dialogue SWT

```
package org.eclipsebook.ch08;

import org.eclipse.swt.*;
import org.eclipse.swt.widgets.*;

public class Ch08_05 {

        public static void main(String [] args) {
            Display display = new Display();
            Shell shell = new Shell(display);
            shell.setSize(280, 200);
            shell.setText("Dialogues");
            shell.open();

            final Button opener = new Button(shell, SWT.PUSH);
            opener.setText("Cliquez Moi");
            opener.setBounds(20, 20, 100, 25);

            final Text text = new Text(shell, SWT.SHADOW_IN);
            text.setBounds(120, 20, 140, 25);

            final Shell dialog = new Shell(shell, SWT.APPLICATION_MODAL |
                SWT.DIALOG_TRIM);
            dialog.setText("Dialogue");
            dialog.setSize(150, 100);

            final Label label = new Label(dialog, SWT.NONE);
            label.setText("OK pour avancer?");
            label.setBounds(35, 5, 100, 20);

            final Button okButton = new Button(dialog, SWT.PUSH);
            okButton.setBounds(20, 35, 40, 25);
            okButton.setText("OK");

            Button cancelButton = new Button(dialog, SWT.PUSH);
            cancelButton.setBounds(70, 35, 60, 25);
            cancelButton.setText("Annuler");

            final boolean [] response = new boolean[1];
            response[0] = true;

            Listener listener = new Listener() {
                 public void handleEvent(Event event) {
                     if(event.widget == okButton){
                         response[0] = true;
                     }else{
                         response[0] = false;
                     }
                     dialog.close();
                 }
            };
```

Exemple 8-5. Création d'un dialogue SWT (suite)

```
            okButton.addListener(SWT.Selection, listener);
            cancelButton.addListener(SWT.Selection, listener);

            Listener openerListener = new Listener() {
                public void handleEvent(Event event) {
                    dialog.open();
                }
            };

            opener.addListener(SWT.Selection, openerListener);

            while(!dialog.isDisposed()) {
                if(!display.readAndDispatch()) display.sleep();
            }

            if(response[0]){
                text.setText("Vous avez cliqué OK");
            } else {
                 text.setText("Vous avez cliqué Annuler");
            }

            while(!shell.isDisposed()) {
                if(!display.readAndDispatch()) display.sleep();
            }
            display.dispose();
        }
    }
```

L'exécution de cet exemple affiche la fenêtre visible dans la figure 8-9 ; un clic sur le bouton permet à l'utilisateur d'afficher le dialogue et le champ texte affichant le résultat.

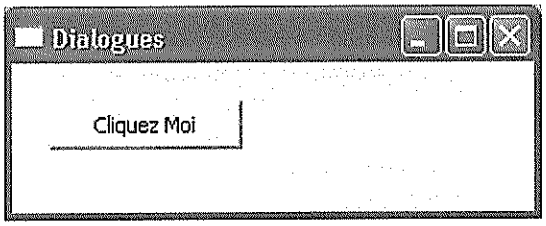


Figure 8-9. L'application Ch08_05

Lorsqu'on clique sur le bouton, la boîte de dialogue apparaît, comme on le voit dans la figure 8-10. Tel qu'elle a été créée, cette boîte ne possède pas de boutons de redimensionnement (minimize et maximize) et ses bords ne sont pas extensibles. Le bouton OK est en surimpression car il s'agit du bouton par défaut ainsi, si l'utilisateur ferme la boîte de dialogue ou presse la touche Entrée, ce bouton OK est sélectionné par défaut.

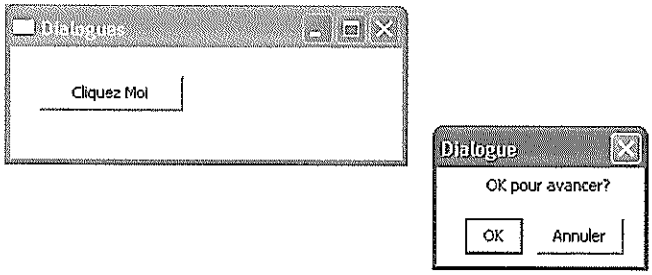


Figure 8-10. Affichage d'une boîte de dialogue

En cliquant sur l'un des boutons, on ferme la boîte de dialogue et l'on affiche, dans le champ texte de la fenêtre principale, le bouton qui a été cliqué (voir la figure 8-11). Ainsi, il a été possible de communiquer entre fenêtres, de récupérer la réponse de l'utilisateur et de l'afficher.

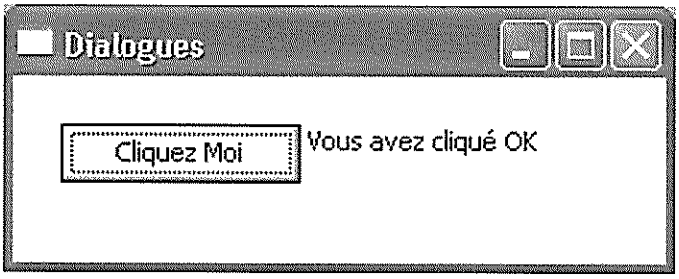


Figure 8-11. Récupération du résultat provenant de la boîte de dialogue

Ouverture d'Internet Explorer dans une fenêtre SWT

Comme on vient de le voir, SWT couvre largement les bases du développement d'interfaces graphiques, offrant tous les contrôles que l'on trouve dans les GUI standard. On peut bien sûr, sans beaucoup d'imagination, se contenter d'interfaces graphiques composées de contrôles standard tels que les curseurs, les champs texte, etc. Mais a contrario, voici pour terminer, un exemple plus intéressant, montrant comment on ouvre Internet Explorer à l'*intérieur* d'une fenêtre SWT et comment on navigue sur une URL sélectionnée.

Par exemple, un utilisateur peut vouloir chercher une information sur Internet, auquel cas cas il ouvre un moteur de recherche. Dans l'exemple suivant, on suppose qu'il recherche un excellent livre d'informatique : il va donc consulter le site *http://www.oreilly.com*.

Dans ce nouveau projet, `Ch08_06`, on va utiliser Object Linking and Embedding (OLE) pour ouvrir Internet Explorer dans une fenêtre SWT ; de fait, cet exemple ne fonctionne que dans l'environnement Windows. Malheureusement, il n'existe pas d'équivalent sous Linux ou Mac OS X – pas encore. Cela étant, il est très pertinent d'observer comment opérer sous Windows.

Par ailleurs, il est *prévu* que dans la version 3.0 de Eclipse soit intégré un élément de navigation (browser widget) – on se reportera à la discussion du chapitre 13.

Pour travailler avec OLE, il faut importer le paquetage `org.eclipse.swt.ole.win32.*` et créer un objet SWT `OleControlSite` dans la méthode `main` :

```
package org.eclipsebook.ch08;

import org.eclipse.swt.*;
import org.eclipse.swt.layout.*;
import org.eclipse.swt.widgets.*;
import org.eclipse.swt.ole.win32.*;

public class Ch08_06 {

        public static void main(String[] args) {

                final Display display = new Display();
                Shell shell = new Shell(display);
                shell.setSize(600, 400);
                shell.setLayout(new FillLayout());

                OleControlSite oleControlSite;
        .
        .
        .
```

La fenêtre que l'on crée est un objet SWT `OleFrame`, dans lequel on place Internet Explorer. Pour y parvenir, on adapte l'objet `OleControlSite` pour travailler avec Internet Explorer, on crée un nouvel objet `OleFrame`, puis on exécute la commande OLE `OLE.OLEIVERB_INPLACEACTIVATE` pour démarrer Internet Explorer. On configure aussi un objet `OleAutomation`, nommé `browser`, grâce auquel on peut interagir avec Internet Explorer :

```
OleControlSite oleControlSite;

OleFrame oleFrame = new OleFrame(shell, SWT.NONE);
oleControlSite = new OleControlSite(oleFrame, SWT.NONE, "Shell.Explorer");
oleControlSite.doVerb(OLE.OLEIVERB_INPLACEACTIVATE);
shell.open();

final OleAutomation browser = new OleAutomation(oleControlSite);
```

Pour travailler avec l'objet `browser`, on peut utiliser la méthode `getIDsOfNames` qui renvoie l'indicatif OLE (OLE IDs) utile à la commande de navigation puis on invoque simplement cette commande pour naviguer sur *http://www.oreilly.com*, comme on peut le voir dans l'exemple 8-6.

Exemple 8-6. Ouverture d'Internet Explorer, Ch08_06.java

```
package org.eclipsebook.ch08;

import org.eclipse.swt.*;
```

Exemple 8-6. Ouverture d'Internet Explorer, Ch08_06.java (suite)

```
import org.eclipse.swt.layout.*;
import org.eclipse.swt.widgets.*;
import org.eclipse.swt.ole.win32.*;

public class Ch08_06 {

    public static void main(String[] args) {

        final Display display = new Display();
        Shell shell = new Shell(display);
        shell.setSize(600, 400);
        shell.setLayout(new FillLayout());

        OleControlSite oleControlSite;

        OleFrame oleFrame = new OleFrame(shell, SWT.NONE);
        oleControlSite = new OleControlSite(oleFrame, SWT.NONE,
                        "Shell.Explorer");
        oleControlSite.doVerb(OLE.OLEIVERB_INPLACEACTIVATE);
        shell.open();

        final OleAutomation browser = new OleAutomation(oleControlSite);

        int[] browserIDs = browser.getIDsOfNames(new String[]{"Navigate",
                        "URL"});
        Variant[] address = new Variant[] {new
                        Variant("http://www.oreilly.com")};
        browser.invoke(browserIDs[0], address, new int[]{browserIDs[1]});

        while (!shell.isDisposed()) {
            if (!display.readAndDispatch())
                display.sleep();
        }

        browser.dispose();
        display.dispose();
    }
}
```

On peut observer le résultat dans la figure 8-12, dans laquelle Internet Explorer a été ouvert pour naviguer sur le site *http://www.oreilly.com*.

Cet exemple est terminé et marque la fin de notre tour d'horizon de SWT. Nombreux sont les sujets qui restent à traiter, qui n'ont pas été abordés dans cet ouvrage. Néanmoins, ces deux chapitres permettent une bonne prise en main de SWT et donnent les techniques nécessaires pour devenir un expert de ce sujet.

La technique générale est assez simple : on crée un objet `Display` et les objets `Shell` pour les différentes fenêtres que l'on veut afficher, on remplit ces fenêtres avec des contrôles, on affiche les fenêtres et l'on définit la boucle de capture des événements. SWT est un outil de développement d'interfaces graphiques très puissant, qui connaît

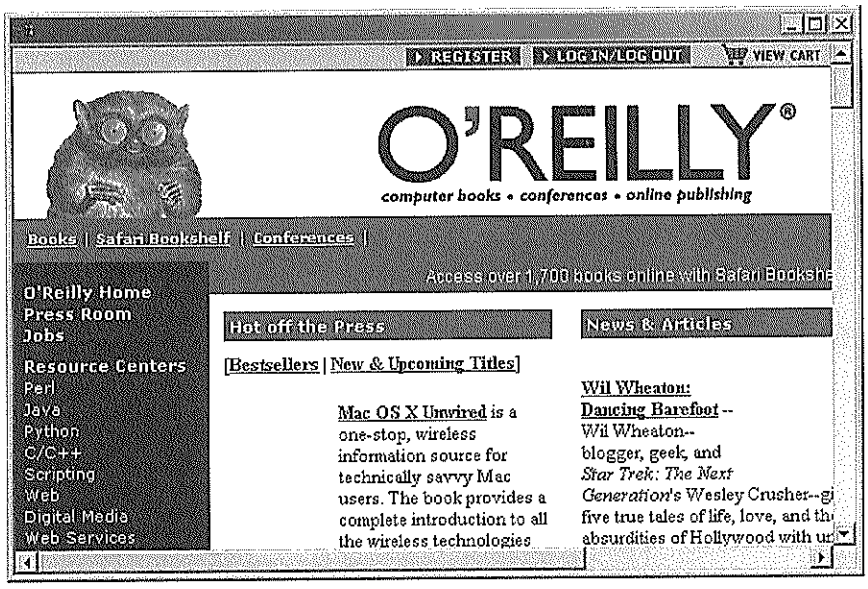


Figure 8-12. Utilisation d'Internet Explorer dans une fenêtre SWT

un succès grandissant. Pour découvrir les contrôles individuels et pour plus de détail sur leur fonctionnement, on se reportera à la documentation livrée avec Eclipse.

CHAPITRE 9

Développement d'applications pour le Web

Java s'est imposé dans le monde du Web en proposant les JSP (JavaServer Pages) ainsi que des outils de développement de servlets. Ce chapitre a pour but de présenter les méthodes de création de servlets et de JSP au moyen d'Eclipse. À cette fin, on utilisera le serveur d'applications Tomcat, qui constitue l'implémentation de référence proposée par Sun Microsystems pour les servlets et les JSP – cet outil peut être téléchargé gratuitement.

Même si Tomcat est utilisé dans ce chapitre, les codes Java et les fichiers XML de ce chapitre ne sont pas spécifiques de Tomcat. Les servlets et les JSP doivent tout de même se conformer à leurs spécifications respectives ; en d'autres termes, tous les développements présentés dans ce chapitre peuvent être repris tels quels dans d'autres conteneurs web de type JSP/servlets.

Installation et configuration de Tomcat

On peut se procurer le serveur d'applications Tomcat sur le site *http://jakarta.apache.org/tomcat/* ; la dernière version de ce produit est la version 4.1.30. Le téléchargement et l'installation de Tomcat sont très simples – il suffit d'utiliser unzip ou untar, ce qui provoque la création de l'arborescence de répertoires suivante :

```
jakarta-tomcat-4.1.30
|_ _bin                    Fichiers binaires exécutables
|_ _common                 Classes utilisables par les classes internes et
                           les applications Web
|   |_ _classes            Classes Java communes
|   |_ _endorsed           Classes Java associées
|   |_ _lib                Classes Java communes au format JAR
|_ _conf                   Fichiers de configuration (mot de passe,...)
|_ _logs                   Fichiers de login du serveur
|_ _server                 Classes Tomcat internes
|_ _shared                 Fichiers partagés
|_ _temp                   Fichiers temporaires
|_ _webapps                Répertoire pour les applications Web
|_ _work                   Répertoire contenant les fichiers temporaires
```

Pour les développeurs d'applications web, le répertoire le plus important est le répertoire *webapps*, dans lequel sont stockés les fichiers destinés à être accessibles par les explorateurs clients. Ce répertoire sera très souvent utilisé tout au long de ce chapitre.

Afin de développer des applications pour le web, on commencera d'abord par utiliser Tomcat en dehors d'Eclipse. On verra ensuite comment lancer Tomcat sous l'environnement Eclipse. La première étape est de s'assurer du bon fonctionnement de Tomcat. Pour cela, il convient d'initialiser les deux variables d'environnement suivantes avant de lancer Tomcat en ligne de commande :

`JAVA_HOME`
: Cette variable doit contenir le répertoire d'installation de Java, qui est le répertoire parent du répertoire Java nommé *bin*. Par exemple, cela pourrait être *c:\jdk1.4* sous Windows.

`CATALINA_HOME`
: Cette variable doit contenir le répertoire d'installation de Tomcat, le répertoire parent du répertoire Tomcat nommé *bin*. Par exemple, cela pourrait être */usr/local/jakarta-tomcat-4.1.30* sous Linux.

Il est possible d'initialiser ces variables d'environnement à partir de la ligne de commande, en tapant par exemple : `set JAVA_HOME=C:\jdk1.4` (sous le shell Unix `tcsh`, il convient d'utiliser plutôt `setenv`). Sur la plupart des systèmes d'exploitation, l'initialisation des variables d'environnement peut être effectuée de façon plus permanente au moyen de boîtes de dialogue de configurations prédéfinies.

Le lecteur trouvera toutes les informations nécessaires à l'initialisation de variables d'environnement sur tous les systèmes d'exploitation acceptant Java en téléchargeant la page Java contenant les notes d'installation. On trouvera les liens vers divers systèmes d'exploitation sur l'URL *http://java.sun.com/j2se/1.4/install.html* (par exemple, l'URL décrivant Windows est *http://java.sun.com/j2se/1.4/install-windows.html*). Ces notes décrivent toutes l'initialisation de la variable d'environnement `PATH`, mais les indications données permettent d'initialiser toute autre variable d'environnement.

Après avoir initialisé ces deux variables d'environnement, il est possible de lancer Tomcat en se plaçant dans le répertoire *bin* de Tomcat et en tapant `startup` sous Windows, ou en lançant `startup.sh` sous Unix. Sous Windows, une nouvelle fenêtre DOS s'ouvre, qui affiche des messages d'initialisation. Pour arrêter Tomcat, il suffit de taper `shutdown` sous Windows ou de lancer `shutdown.sh` sous Unix.

Le lecteur trouvera des indications spécifiques pour le lancement de Tomcat dans le document *running.txt* qui est fourni avec Tomcat.

Maintenant que Tomcat fonctionne, on peut lancer un explorateur et accéder à *http://localhost:8080*, ce qui doit ouvrir la page de bienvenue de Tomcat (voir figure 9-1). La partie « localhost » de cette URL est celle qui est utilisée pour des serveurs web locaux (elle correspond à l'adresse IP `127.0.0.1`) ; 8080 est le numéro de port ; les serveurs

web utilisent généralement le port 80, mais Tomcat utilise le port 8080 pour éviter les conflits avec d'autres serveurs.

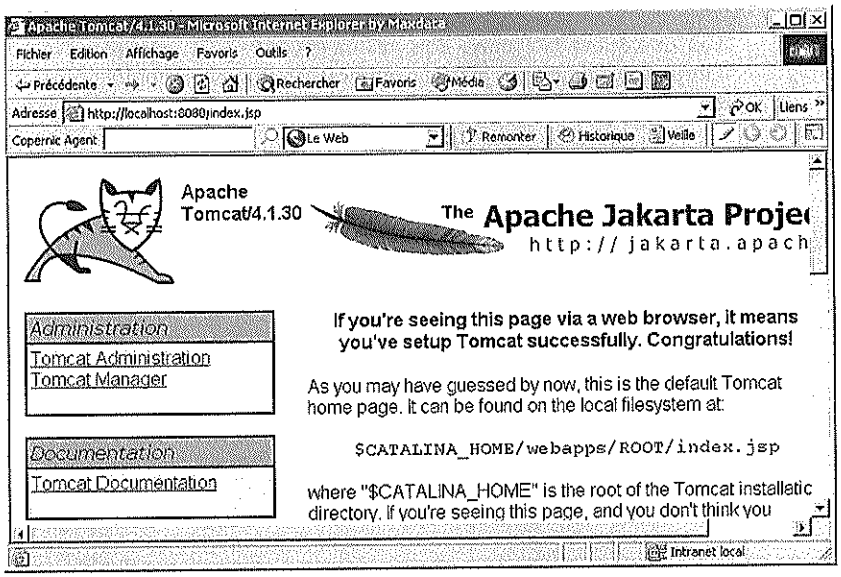


Figure 9-1. La page de bienvenue de Tomcat

Création d'une JSP

Le serveur web fonctionne, et il est grand temps de l'utiliser. L'utilisation la plus simple consiste à employer Eclipse pour écrire des fichiers JSP, ce qui ne nécessite aucune compilation. Ces fichiers peuvent contenir du code Java dans des éléments de type *scriptlet*, *declaration* ou *expression*. Les éléments les plus généraux sont les scriptlets, qui peuvent contenir plusieurs lignes de code Java. Les scriptlets sont délimitées par les balises <% et %>, comme on peut le voir dans l'exemple 9-1. La méthode `println` de l'objet `out` peut être utilisée pour envoyer du texte à l'explorateur ; dans l'exemple ci-dessous, le texte « Utilisation de JSP » est envoyé à l'explorateur par la JSP.

Exemple 9-1. Un exemple de JSP

```
<HTML>
  <HEAD>
    <TITLE>Une page web</TITLE>
  </HEAD>

  <BODY>
    <H1>Au travail avec JSP</H1>
    <% out.println("Utilisation de JSP"); %>
  </BODY>
</HTML>
```

On peut très simplement créer un fichier JSP en le saisissant sous Eclipse, comme on peut le voir dans la figure 9-2 : dans celle-ci, on a créé un nouveau projet, nommé `Ch09_01`, ainsi qu'un nouveau fichier, nommé *Ch09_01.jsp*, qui va recevoir le code

JSP. Il n'y a aucune vérification de syntaxe à ce niveau ; Eclipse utilise simplement son éditeur standard par défaut.

Afin de vérifier la syntaxe d'un document JSP, il est possible d'utiliser l'éditeur XML gratuit nommé XML Buddy, disponible à l'adresse *http://www.xmlbuddy.com*.

Une fois cette étape franchie, le problème consiste à installer *Ch09_01.jsp* de telle

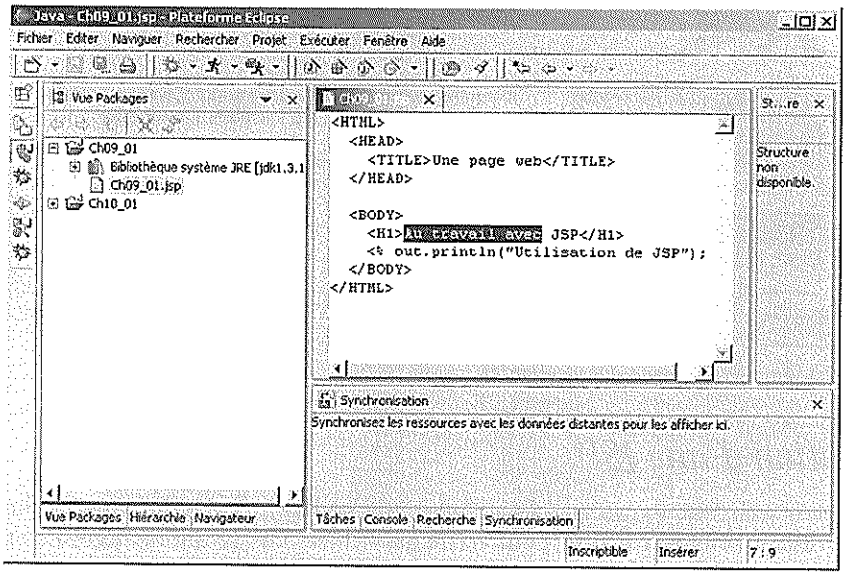


Figure 9-2. Saisie de code JSP

façon que Tomcat puisse le fournir aux explorateurs clients. Afin de mettre en œuvre les exemples de ce chapitre, on va créer un sous-répertoire du répertoire Tomcat *webapps*. Ce nouveau répertoire, appelé ici *Ch09*, doit lui-même contenir un sous-répertoire nommé *WEB-INF*, qui doit contenir deux sous-sous-répertoires nommés *classes* et *lib* :

```
webapps
|_ _Ch09                  Répertoire contenant les exemples du chapitre 9
    |_ _WEB-INF           Informations sur les applications Web du chapitre 9
        |_ _classes       Classes Java utilisées par les applications Web du
                          chapitre 9
        |_ _lib           fichiers JAR utilisés par les applications Web du
                          chapitre 9
```

Les répertoires *WEB-INF*, *classes* et *lib* sont actuellement vides - ils doivent néanmoins exister, sinon Tomcat considérera que le répertoire *Ch09* est non valide pour stocker des documents web. Arrivé à cette étape, il faut stocker *Ch09_01.jsp* dans le répertoire *Ch09*, puis il faut fermer Tomcat (s'il tourne) et relancer son exécution.

Lorsqu'on ajoute un nouveau sous-répertoire dans le répertoire *webapps*, ou lorsqu'on installe des fichiers *.class* où que ce soit dans ce même répertoire *webapps*, il faut absolument fermer Tomcat puis le relancer. Par défaut, Tomcat copie le contenu du répertoire *webapps* dans le répertoire *work*, qui constitue le répertoire de travail réel utilisé par la suite par Tomcat. Il est donc nécessaire de fermer Tomcat puis de le relancer après toute modification majeure du répertoire *webapps*. Tomcat peut être configuré pour détecter ces changements, afin qu'il ne soit pas nécessaire de le relancer, mais cette possibilité ne sera pas utilisée ici.

Afin de voir le document JSP au travail, on peut naviguer dans *http://localhost:8080/Ch09/Ch09_01.jsp*, comme l'illustre la figure 9-3. Tomcat traduit le code Java contenu dans la JSP sous la forme d'un servlet, qui est compilé puis exécuté : la figure en montre le résultat.

Figure 9-3. Visualisation d'un fichier JSP

On vient ainsi de créer une première JSP au moyen de Tomcat, mais, comme on peut le constater, Eclipse n'a pas été beaucoup employé. En fait, Eclipse n'a été utilisé que pour créer *Ch09_01.jsp*. Il est donc grand temps de l'utiliser réellement.

Création d'un servlet

Les JSP ont été introduites afin de rendre la programmation Java pour le web plus accessible aux novices, leur permettant ainsi de mélanger du code HTML et du code Java. En réalité, les JSP, qui sont basées sur les servlets Java (elles sont traduites en servlets avant leur exécution), constituent du pur code Java, et la plus grande partie de ce chapitre leur est consacrée. L'exemple 9-2 présente un exemple de servlet.

Exemple 9-2. Un exemple de servlet

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
```

Exemple 9-2. Un exemple de servlet (suite)

```
public class ch09_02 extends HttpServlet
{
    public void doGet(HttpServletRequest request,
        HttpServletResponse response)
        throws IOException, ServletException
    {
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();

        out.println("<HTML>");
        out.println("<HEAD>");
        out.println("<TITLE>");
        out.println("Un exemple de servlet");
        out.println("</TITLE>");
        out.println("</HEAD>");
        out.println("<BODY>");
        out.println("<H1>");
        out.println("Travailler avec les servlets");
        out.println("</H1>");
        out.println("Utiliser les servlets");
        out.println("</BODY>");
        out.println("</HTML>");
    }
}
```

Les servlets tels que celui de l'exemple ci-dessus sont basés sur la classe `javax.servlet.http.HttpServlet`, et ils se contentent souvent de surcharger la méthode `doGet`, qui reçoit l'objet `request` détenant les données en provenance de l'explorateur (y compris le type d'explorateur ainsi que les données associées à tout contrôle HTML) ainsi que l'objet `response` qui permet au programmeur de mettre en forme sa réponse à l'explorateur client.

On surcharge la méthode `doGet` pour gérer les requêtes HTTP `GET` (par exemple, lorsque l'attribut `METHOD` d'un formulaire HTML a pour valeur `GET`) ou les accès servlet par défaut. La méthode `doPost`, qui possède les mêmes arguments, gère la méthode `POST` (par exemple, lorsque l'attribut `METHOD` d'un formulaire HTML a pour valeur `POST`). Pour gérer les requêtes `GET` ou `POST`, il est possible de surcharger la méthode `service`, qui possède aussi la même liste d'arguments (et qui est en réalité la méthode responsable de l'appel de `doGet` ou de `doPost`).

Dans l'exemple présent, on élabore une réponse formatée à l'explorateur en utilisant la méthode `getWriter` de l'objet `response` pour obtenir une instance de la classe `PrintWriter` ; la méthode `println` de cette instance permet de faire parvenir un document au format HTML à l'explorateur. Ici, on se contente d'envoyer une page HTML contenant le texte « Utiliser les servlets ».

On poursuit en créant un nouveau projet, nommé `Ch09_02`, dont on saisit le code dans la nouvelle classe `Ch09_02` (cette classe, stockée dans le fichier *Ch09_02.java*, fait

partie du paquetage *org.eclipsebook.ch09*). Ce code requiert l'importation de *servlet.jar*, qui est fourni avec Tomcat, dans le chemin de génération. L'emplacement du fichier *servlet.jar* est en général *jakarta-tomcat-4.1.30\common\lib\servlet.jar* ; pour cela, il suffit de cliquer le bouton droit de la souris sur le projet Ch09_02 dans la Vue Packages, de sélectionner Propriétés et d'ajouter *servlet.jar* au chemin de compilation Java (onglet Bibliothèques, bouton Ajouter des fichiers JAR externes).

On peut enfin construire le servlet en sélectionnant le projet Ch09_02, dans la Vue Packages, et en sélectionnant Projet → Recompiler un projet : le fichier *Ch09_02.class* est alors créé. Les fichiers class tels que celui-ci vont dans le répertoire Tomcat *classes*. Notre servlet se trouve dans le paquetage `org.eclipsebook.ch09`, et comme la structure des répertoires doit dupliquer celle des paquetages, il faut placer *Ch09_02.class* dans *webapps\Ch09\WEB-INF\class\org\eclipsebook\ch09*:

```
webapps
|_ _ch09
    |_ _WEB-INF
         |_ _classes
         |   |_ _org
         |       |_ _eclipsebook
         |           |_ _ch09  Répertoire recevant le code du servlet
         |_ _lib
```

Pour faire savoir à Tomcat que cette nouvelle classe est un servlet, il faut utiliser un fichier nommé *web.xml*, qui contient des données relatives à la configuration des applications pour le web. Dans ce fichier, on utilise deux éléments XML, `servlet` et `servlet-mapping`, pour connecter l'URL *Ch09_02* au code Java réel du servlet, `org.eclipsebook.ch09.Ch09_02`. On peut voir l'aspect de *web.xml* dans l'exemple 9-3.

Exemple 9-3. Mise à jour de web.xml

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE web-app PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application
                          2.3//EN"
"http://java.sun.com/dtd/web-app_2_3.dtd">
<web-app>
  <display-name>Un exemple d'application</display-name>

  <servlet>
    <servlet-name>Ch09_02</servlet-name>
    <servlet-class>org.eclipsebook.ch09.Ch09_02</servlet-class>
  </servlet>

  <servlet-mapping>
    <servlet-name>Ch09_02</servlet-name>
    <url-pattern>/org.eclipsebook.ch09.Ch09_02</url-pattern>
  </servlet-mapping>

</web-app>
```

On crée donc le nouveau document *web.xml* sous Eclipse, en cliquant le bouton droit de la souris sur le projet et en sélectionnant Nouveau → Fichier. Pour ouvrir *web.xml* sous Eclipse, il faut cliquer ce même bouton droit et sélectionner l'option Ouvrir avec → Editeur de texte (si l'on double-clique sur le fichier, c'est l'éditeur XML par défaut

– par exemple, l'Explorateur Internet de Windows – qui sera ouvert, sauf si l'on dispose d'un plug-in XML tel que XMLBuddy). On tape le code XML de l'exemple 9-3, on sauvegarde le fichier, puis on le copie dans le sous-répertoire *WEB-INF* du répertoire *Ch09* :

```
webapps
|_ _ch09
    |_ _WEB-INF          Informations sur les applications web du chapitre 9
        |_ _classes
        |_ _lib
```

On peut enfin fermer Tomcat puis le relancer : tout est prêt. Sous l'explorateur, on navigue sur l'URL *http://localhost:8080/Ch09/org.eclipsebook.ch09.Ch09_02* : on doit obtenir le résultat de la figure 9-4.

Figure 9-4. Visualisation d'un nouveau servlet

Dans cet exemple, on a produit du code et on l'a copié dans les répertoires Tomcat, mais Eclipse peut se charger de la gestion des fichiers à la place du programmeur.

Création d'un servlet avec gestion automatique des répertoires

Le code développé doit se trouver dans les bons répertoires Tomcat : Eclipse peut se charger de placer les fichiers au bon endroit. Pour le voir, on peut créer le nouveau projet `Ch09_03`. On tape le nom du projet dans la boîte de dialogue Nouveau projet Java et on clique sur Suivant pour accéder à la deuxième page de celle-ci. Pour redéfinir le répertoire de sortie recevant le produit des compilations Java, il faut cliquer le bouton Parcourir se trouvant face à la zone de texte modifiable intitulée Dossier de sortie par défaut, ce qui ouvre la boîte de dialogue Sélection du dossier. On clique sur le bouton Créer un dossier, puis sur le bouton Avancées, dans la boîte de dialogue Nouveau dossier. On coche ensuite la case Lier au dossier dans le système de fichiers, puis on clique sur le bouton Parcourir. On sélectionne le répertoire *jakarta-tomcat-4.1.30\webapps\Ch09\WEB-INF\classes* – la racine du code compilé du servlet – puis on clique OK, revenant ainsi à la boîte de dialogue Nouveau dossier. Dans ce répertoire, on nomme ce nouveau dossier output (voir figure 9-5), et enfin on clique OK.

Réinitialiser ainsi le répertoire de sortie fait que le code compilé sera automatiquement placé dans le sous-répertoire *classes* (ou, dans le cas présent, dans

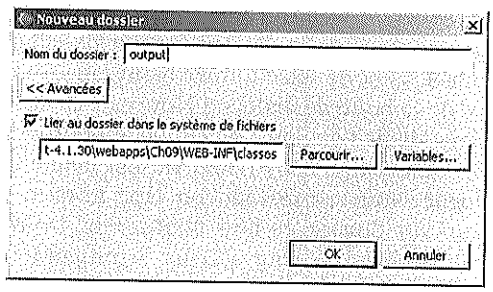


Figure 9-5. Création d'un nouveau répertoire de sortie

classes\org\eclipsebook\ch09, en vertu du nom du paquetage utilisé). On peut alors créer la nouvelle classe `Ch09_03` et taper le code du servlet de l'exemple 9-4.

Exemple 9-4. Un nouveau servlet

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class Ch09_03 extends HttpServlet
{
    public void doGet(HttpServletRequest request,
        HttpServletResponse response)
        throws IOException, ServletException
    {
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();

        out.println("<HTML>");
        out.println("<HEAD>");
        out.println("<TITLE>");
        out.println("Un exemple de servlet");
        out.println("</TITLE>");
        out.println("</HEAD>");
        out.println("<BODY>");
        out.println("<H1>");
        out.println("Travailler avec les servlets");
        out.println("</H1>");
        out.println("Développer des servlets avec gestion automatique des
                    répertoires");
        out.println("</BODY>");
        out.println("</HTML>");
    }
}
```

Ajoutons enfin *servlet.jar* au chemin de compilation comme précédemment. Lorsque le projet sera construit, *Ch09_03.class* sera automatiquement stocké dans le répertoire Tomcat *webapps\Ch09\WEB-INF\classes\org\eclipsebook\ch09*, comme attendu.

Pour installer le nouveau servlet, il faut aussi éditer *web.xml*, ce qui peut être effectué sous Eclipse au moyen d'un répertoire lié (*linked folder*). Dans l'exemple présent, on

va lier le répertoire *Ch09/WEB-INF* du répertoire d'installation de Tomcat, qui contient *web.xml*. Pour créer un répertoire lié, il suffit de cliquer le bouton droit de la souris sur le projet `Ch09_03` et de sélectionner Nouveau → Dossier. On clique alors sur le bouton Avancées de la boîte de dialogue Nouveau dossier, on coche la case Lier au dossier dans le système de fichiers, puis on clique le bouton Parcourir et l'on sélectionne *jakarta-tomcat-4.1.30\webapps\Ch09\WEB-INF*. On clique ensuite OK pour revenir à la boîte de dialogue Nouveau dossier, on tape le nom `WEB-INF` dans la zone d'édition intitulée Nom du dossier et l'on clique Fin. Le nouveau répertoire lié nommé *WEB-INF* est alors créé : on peut accéder à son contenu, et en particulier à *web.xml*, en l'ouvrant dans la Vue Packages, comme on peut le voir dans la figure 9-6.

Le fichier *web.xml* à utiliser dans ce projet apparaît dans l'exemple 9-5 (on notera que

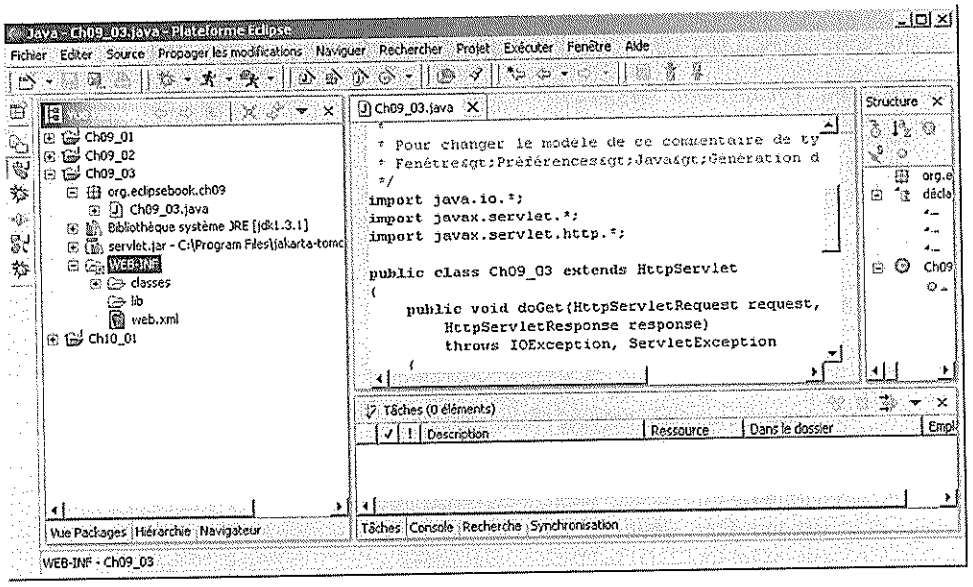


Figure 9-6. Utilisation d'un répertoire lié

les éléments `servlet` doivent être groupés ensemble et qu'ils doivent être suivis par les éléments `servlet-mapping` eux aussi groupés ensemble – ces éléments ne doivent pas être mélangés, car sinon Tomcat ne pourrait pas décoder convenablement le fichier et ne pourrait donc pas démarrer).

Exemple 9-5. La nouvelle version de web.xml

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE web-app PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application
                          2.3//EN"
"http://java.sun.com/dtd/web-app_2_3.dtd">
<web-app>
  <display-name>Exemples d'applications</display-name>

  <servlet>
    <servlet-name>Ch09_02</servlet-name>
```

Exemple 9-5. La nouvelle version de web.xml (suite)

```
    <servlet-class>org.eclipsebook.ch09.Ch09_02</servlet-class>
  </servlet>

  <servlet>
    <servlet-name>Ch09_03</servlet-name>
    <servlet-class>org.eclipsebook.ch09.Ch09_03</servlet-class>
  </servlet>

  <servlet-mapping>
    <servlet-name>Ch09_02</servlet-name>
    <url-pattern>/org.eclipsebook.ch09.Ch09_02</url-pattern>
  </servlet-mapping>

  <servlet-mapping>
    <servlet-name>Ch09_03</servlet-name>
    <url-pattern>/org.eclipsebook.ch09.Ch09_03</url-pattern>
  </servlet-mapping>

</web-app>
```

On tape cette nouvelle version de *web.xml* et on relance Tomcat. Eclipse se charge de la bonne gestion des fichiers : il n'y a donc rien à copier dans les répertoires Tomcat – on peut maintenant naviguer sur l'URL du nouveau servlet, *http://localhost:8080/Ch09/org.eclipsebook.ch09.Ch09_03* (voir figure 9-7).

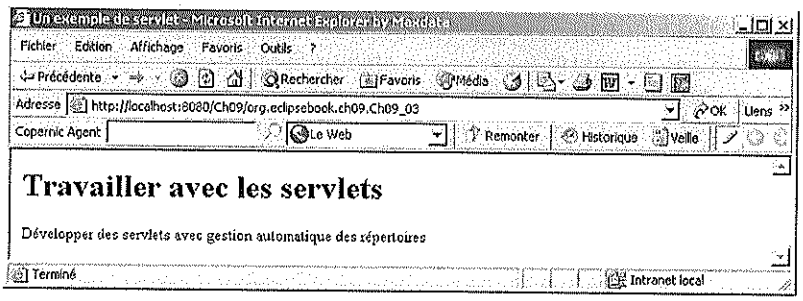


Figure 9-7. Développement d'un servlet avec gestion automatique des répertoires

Cette fois, tout le code du servlet a été développé sous Eclipse, qui a même géré les fichiers lui-même.

Connexion à un JavaBean

Les fichiers JSP sont capables de se connecter à du code Java compilé au moyen de JavaBeans ; le développement de telles applications ne pose aucun problème si l'on utilise les répertoires liés. Le projet `Ch09_04` en est un exemple. Dans celui-ci on utilise le bean de l'exemple 9-6, qui possède une propriété nommée `msg` qui retourne le message « Bonjour chez vous. »

Exemple 9-6. Un JavaBean

```
package org.eclipsebook.ch09;

public class Ch09_04 {
        private String message = "Bonjour chez vous.";

        public void setMessage(String msg)
        {
                this.message = msg;
        }

        public String getMessage()
        {
                return this.message;
        }

        public Ch09_04()
        {
        }
}
```

Le fichier *Ch09_04.class*, obtenu grâce à la compilation de ce code, est destiné à se trouver dans le répertoire *webapps\Ch09\WEB-INF\classes*, qui sera donc le répertoire de sortie (on note que le fichier *servlet.jar* n'est pas nécessaire dans le chemin de compilation). Une fois le code tapé, il faut donc le compiler pour créer *Ch09_04.class*.

Sous JSP, on peut connecter du code bean de nombreuses façons ; par exemple, on peut instancier la classe `Ch09_04`, grâce à Java, dans un scriptlet. Cependant, il est recommandé d'utiliser l'élément JSP `jsp:useBean` pour créer un objet JavaBean, puis d'utiliser l'élément `jsp:getProperty` pour obtenir la valeur d'une propriété de bean, ainsi que `jsp:setProperty` pour modifier cette valeur. L'exemple 9-7 illustre la mise en œuvre de tout cela : la valeur courante de la propriété `msg` du bean y est lue, puis modifiée, cette nouvelle valeur étant lue à son tour dans le bean.

Exemple 9-7. Connexion à un JavaBean

```
<HTML>
    <HEAD>
        <TITLE>Modification de la valeur d'une propriété</TITLE>
    </HEAD>

    <BODY>
        <H1>Modification de la valeur d'une propriété</H1>

        <jsp:useBean id="bean1" class="org.eclipsebook.ch09.Ch09_04" />

        Le message est: <jsp:getProperty name="bean1" property="message" />
        <BR>
        <jsp:setProperty name="bean1" property="message" value="Coucou !" />

        Maintenant le message est: <jsp:getProperty name="bean1"
                        property="message" />
    </BODY>
</HTML>
```

Exemple 9-7. Connexion à un JavaBean (suite)

Cette JSP affiche le message original contenu dans la propriété msg ainsi que la nouvelle valeur de celle-ci, après sa modification. Grâce au répertoire lié, on peut stocker ce nouveau fichier JSP dans le répertoire *webapps\Ch09* et naviguer sur *http://localhost:8080/Ch09/Ch09_04.jsp* (voir figure 9-8).

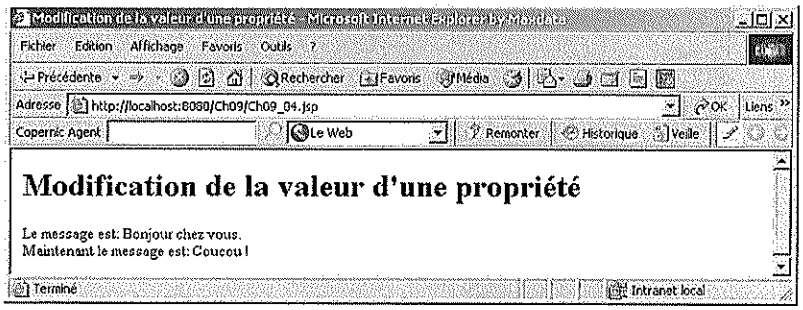


Figure 9-8. Connexion à un JavaBean

On vient de développer du code Java sous Eclipse : on l'a compilé et installé dans Tomcat. Les seules actions à effectuer en-dehors d'Eclipse ont été de lancer et d'arrêter Tomcat. En fait, il s'avère possible de réaliser aussi ces deux actions sous Eclipse.

Utilisation du plug-in Tomcat Sysdeo

Le plug-in Sysdeo, que l'on va utiliser ici, permet de lancer et d'arrêter Tomcat à partir d'Eclipse. Celui-ci peut être téléchargé gratuitement à partir de *http://www.sysdeo.com/eclipse/tomcatPlugin.html*. Après l'avoir décompressé dans le répertoire *plugins*, on peut l'activer en sélectionnant l'option Fenêtre → Personnaliser la perspective, qui ouvre le nœud Autre, et en sélectionnant l'élément Tomcat (voir figure 9-9).

Cela ajoute un menu Tomcat à Eclipse (voir figure 9-10) ainsi que trois boutons Tomcat à la barre d'outils Eclipse (visibles sous le menu Propager les modifications) ; ces boutons permettent de lancer, d'arrêter et de relancer Tomcat.

Pour connecter le plug-in Tomcat à la version de Tomcat utilisée sur la plateforme, il suffit de sélectionner Fenêtre → Préférences et de cliquer le nœud Tomcat, comme on peut le voir dans la figure 9-11.

Cette boîte de dialogue permet de connecter le plug-in Tomcat à Tomcat lui-même. Au moment de la rédaction de cet ouvrage, le plug-in Sysdeo pour Eclipse 2.1.1 existait en version bêta et ne fonctionnait pas correctement avec la version la plus récente de Tomcat, la version 4.1.30. C'est pourquoi une version antérieure de Tomcat va être utilisée dans les deux exemples suivants (ces versions antérieures de Tomcat peuvent être extraites des archives Apache sur l'URL *http://archive.apache.org/dist/jakarta/* ; les archives de toutes les versions 4+ de Tomcat se trouvent sur *http://archive.apache.org/dist/jakarta/tomcat-4/archive/*). Il est très problable qu'à la parution de cet ouvrage, le plug-in Tomcat fonctionne correctement avec la dernière version de Tomcat ; si tel n'est pas le cas, il convient de télécharger une version antérieure de Tomcat sur le site Tomcat et d'utiliser celle-ci pour les exemples ci-dessous.

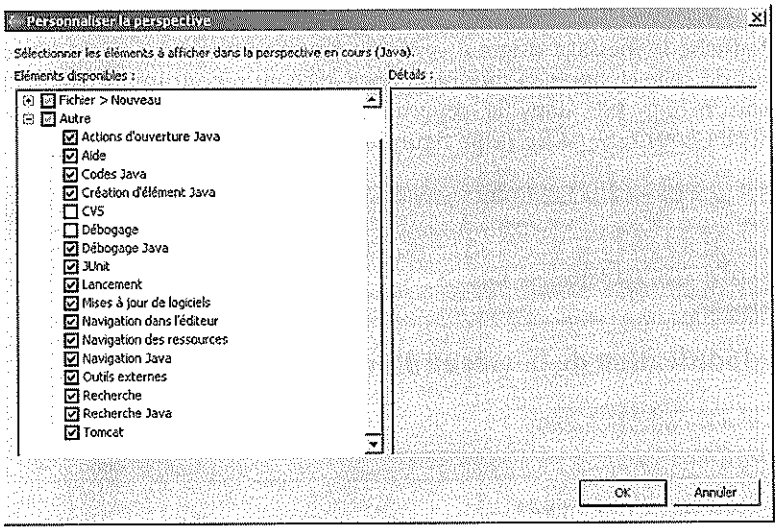


Figure 9-9. Activation du plug-in Tomcat

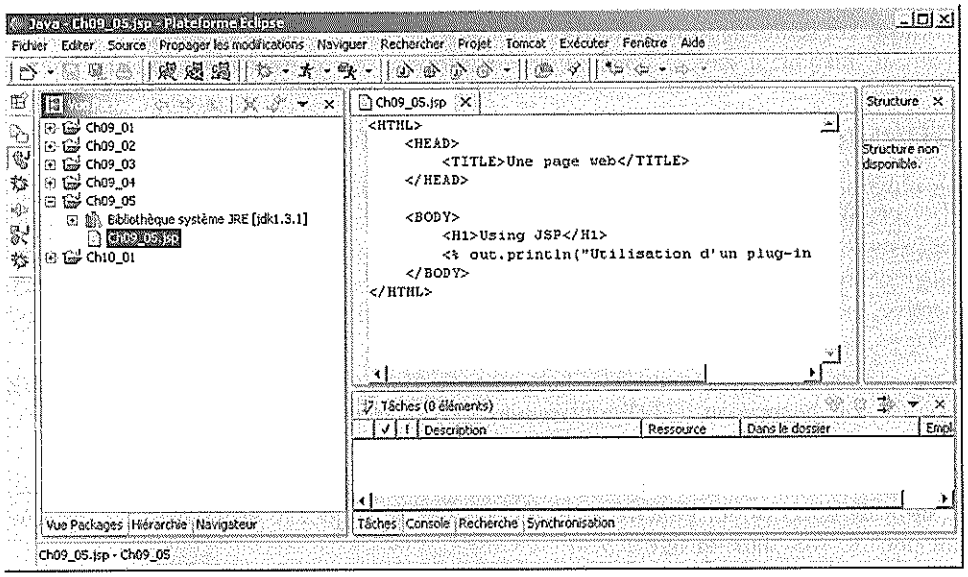


Figure 9-10. Nouveaux boutons Tomcat dans la barre d'outils

Dans cette boîte de dialogue, on initialise le répertoire d'installation de Tomcat, qui est le répertoire parent du répertoire Tomcat *bin* (équivalent à la variable d'environnement `CATALINA_HOME`, présentée plus haut). Pour initialiser cet emplacement, on clique le bouton Parcourir se trouvant face à la zone de saisie Répertoire de Tomcat, on sélectionne le répertoire d'installation de Tomcat et l'on clique OK (ceci permet de remplir automatiquement la zone de saisie Fichier de configuration). On clique OK pour fermer la boîte de dialogue Préférences.

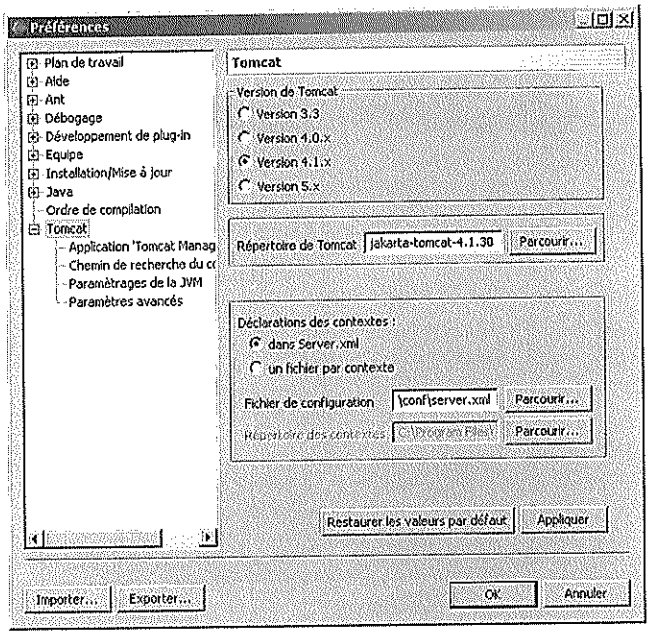


Figure 9-11. Connexion du plug-in Tomcat à Tomcat

Écriture de JSP au moyen du plug-in Tomcat Sysdeo

Nous allons tout d'abord utiliser le plug-in Tomcat dans le cadre d'un projet JSP. Pour créer un nouveau projet Tomcat, il suffit de sélectionner Fichier → Nouveau → Projet puis, dans la boîte de dialogue Nouveau projet, le choix Java dans la partie gauche de la boîte, Projet Tomcat dans sa partie droite (voir figure 9-12), puis on clique Suivant. On nomme ce nouveau projet `Ch09_05` dans la boîte de dialogue Nouveau projet Tomcat et on clique Fin. Le nouveau projet est alors créé, ainsi qu'un certain nombre de répertoires.

Pour créer un nouveau projet JSP, il faut cliquer le bouton droit de la souris sur le projet et sélectionner Nouveau → Fichier : le nouveau fichier à créer s'appelle ici *Ch09_05.jsp*. On tape alors le code JSP de l'exemple 9-8. Une fois ce fichier sauvegardé, il est prêt à être exécuté (il est inutile de le copier).

Exemple 9-8. Création d'une JSP au moyen du plug-in Sysdeo

```
<HTML>
    <HEAD>
        <TITLE>Une page web</TITLE>
    </HEAD>

    <BODY>
        <H1>Utilisation de JSP</H1>
        <% out.println("Utilisation d'un plug-in Tomcat..."); %>
    </BODY>
</HTML>
```

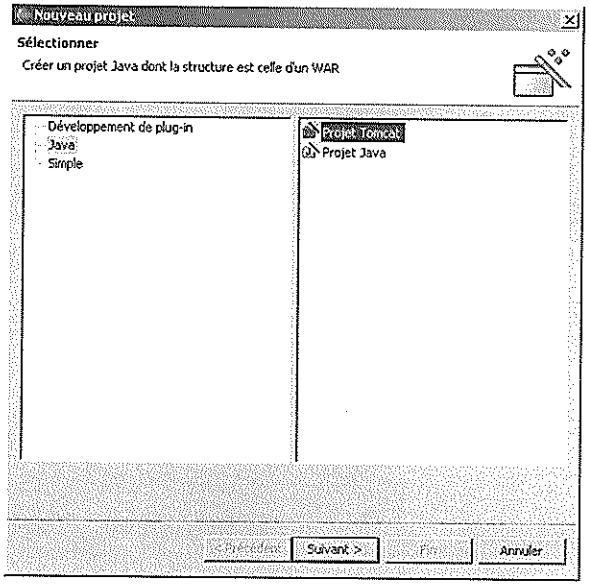


Figure 9-12. Creation d'un nouveau projet Tomcat

Exemple 9-8. Création d'une JSP au moyen du plug-in Sysdeo (suite)

Le plug-in Tomcat permet d'éditer localement des fichiers ; il modifie pour eux le contexte Tomcat au travers du fichier *server.xml* qui se trouve dans le répertoire Tomcat *conf*. Dans l'exemple présent, le plug-in ajoute les lignes suivantes dans *server.xml* pour que Tomcat sache où trouver le fichier JSP :

```
<Context path="/Ch09_05" reloadable="true"
docBase="D:\eclipseworkspace\Ch09_05"
workDir="D:\eclipseworkspace\Ch09_05\work\org\apache\jsp" />
```

Cette modification du fichier *server.xml* permet à Tomcat de trouver la JSP : on peut donc démarrer Tomcat en cliquant sur le bouton Démarrer Tomcat et naviguer sur *http://localhost:8080/Ch09_05/Ch09_05.jsp* au moyen d'un explorateur (cette URL inclut le nom du projet, `Ch09_05`, et pas seulement le répertoire *Ch09*, comme dans les exemples précédents). Le résultat obtenu est visible dans la figure 9-13.

La perspective Débogage va apparaître au lancement de Tomcat ; quand on en aura terminé avec cet exemple, on terminera la session de débogage et on cliquera le bouton Supprime tous les programmes terminés dans la vue Débogage.

Cette JSP tourne. Comme on peut le voir, le plug-in Tomcat permet de développer du code sous Eclipse et aussi de lancer ou d'arrêter Tomcat.

Écriture de servlets avec le plug-in Tomcat Sysdeo

Le plug-in Sysdeo permet aussi de développer des servlets, comme celui de l'exemple 9-9. Jusqu'à maintenant, on a dû initilialiser le répertoire de sortie pour stocker les

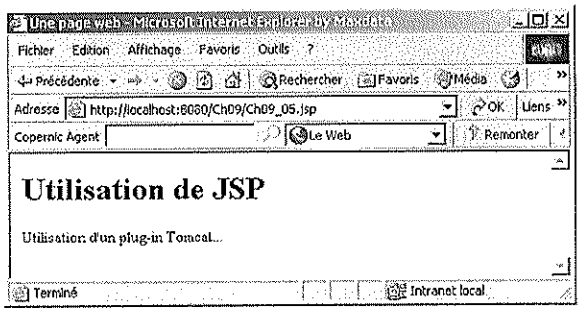


Figure 9-13. Création d'une nouvelle JSP au moyen du plug-in Sysdeo

servlets dans les répertoires Tomcat, mais le plug-in Tomcat peut aussi gérer cela automatiquement.

Exemple 9-9. Création d'un servlet au moyen du plug-in Sysdeo

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class Ch09_06 extends HttpServlet {

                public void doGet(HttpServletRequest request,
                        HttpServletResponse response)
                        throws IOException, ServletException
                {
                        response.setContentType("text/html");
                        PrintWriter out = response.getWriter();

                        out.println("<HTML>");
                        out.println("<HEAD>");
                        out.println("<TITLE>");
                        out.println("Une page web");
                        out.println("</TITLE>");
                        out.println("</HEAD>");
                        out.println("<BODY>");
                        out.println("<H1>");
                        out.println("Utilisation de servlet");
                        out.println("</H1>");
                        out.println("Utilisation d'un plug-in Tomcat...");
                        out.println("</BODY>");
                        out.println("</HTML>");
                }
}
```

Pour voir le fonctionnement du servlet, on crée un nouveau projet Tomcat, `Ch09_06`. On y ajoute une nouvelle classe, `Ch09_06`, en l'incluant dans le paquetage `org.eclipsebook.ch09`. Cette nouvelle classe est stockée dans le répertoire *WEB-INF/src*, comme on peut le voir dans la figure 9-14.

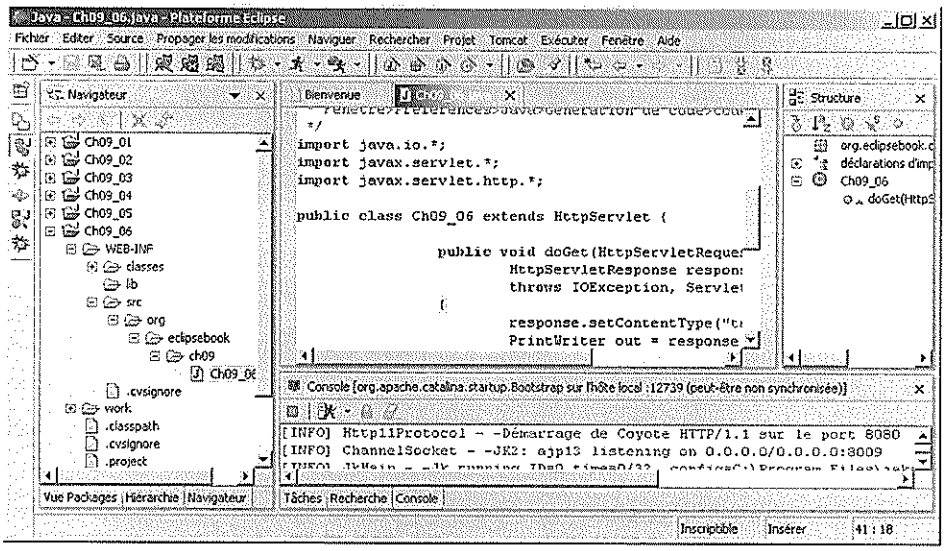


Figure 9-14. Un projet de servlet utilisant le plug-in Tomcat

Comme précédemment, le plug-in Tomcat édite le document Tomcat *server.xml* pour permettre à Tomcat de savoir où trouver les fichiers :

```
<Context path="/Ch09_06" reloadable="true"
docBase="D:\eclipseworkspace\Ch09_06"
workDir="D:\eclipseworkspace\Ch09_06\work\org\apache\jsp" />
```

Il faut réaliser une action supplémentaire avant de lancer ce servlet. Le plug-in Tomcat ne crée pas de fichier *web.xml* local pour faire savoir à Tomcat que *Ch09_06.class* est un servlet, chose faite dans les exemples précédents de ce chapitre. Pour faire en sorte que Tomcat exécute des classes servlet anonymes non définies dans le fichier *web.xml*, il faut activer le servlet « invocateur » de Tomcat en ôtant les commentaires qui entourent l'élément `servlet-mapping` dans le fichier *web.xml* situé dans le répertoire Tomcat *conf* :

```
  <!-- The mapping for the invoker servlet -->
<!--
  <servlet-mapping>
    <servlet-name>invoker</servlet-name>
    <url-pattern>/servlet/*</url-pattern>
  </servlet-mapping>
-->
```

Une fois le fichier *web.xml* du répertoire Tomcat *conf* édité, le servlet est prêt à être exécuté ; on sauvegarde *Ch09_06.java* et l'on construit le projet, puis on lance Tomcat au moyen des boutons du plug-in Tomcat et l'on navigue sur *http://localhost:8080/Ch09_06/servlet/org.eclipsebook.ch09.Ch09_06*. Le résultat apparaît dans la figure 9-15. Voilà un servlet créé grâce au plug-in Tomcat Sysdeo.

De façon alternative, si le programmeur ne veut pas éditer le fichier *web.xml* du répertoire Tomcat *conf* pour activer les servlets anonymes, il peut créer une version locale de *web.xml* pour chaque projet qu'il crée avec le plug-in Tomcat. Pour ce faire,

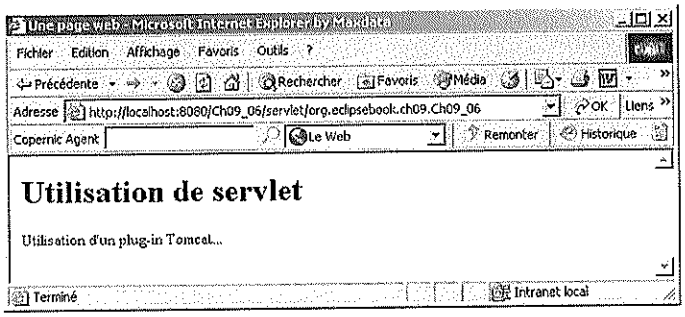


Figure 9-15. Exécution d'un servlet au moyen du plug-in Tomcat

il suffit de cliquer le bouton droit de la souris sur le dossier *WEB-INF*, dans la Vue Packages, de sélectionner Nouveau → Fichier, et de taper le code XML de l'exemple 9-10.

Exemple 9-10. Un fichier web.xml à utiliser avec le plug-in Sysdeo

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE web-app PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application
                          2.3//EN" "http://
java.sun.com/dtd/web-app_2_3.dtd">
<web-app>
  <display-name>Exemple d'application</display-name>

  <servlet>
    <servlet-name>Ch09_06</servlet-name>
    <servlet-class>org.eclipsebook.ch09.Ch09_06</servlet-class>
  </servlet>

  <servlet-mapping>
    <servlet-name>Ch09_06</servlet-name>
    <url-pattern>/org.eclipsebook.ch09.Ch09_06</url-pattern>
  </servlet-mapping>

</web-app>
```

Si une version locale du fichier *web.xml* telle que celle-ci est créée pour chaque projet, le mot « servlet » est inutile dans l'URL du servlet – dans ce cas, on peut naviguer sur le servlet d'URL *http://localhost:8080/Ch09_06/org.eclipsebook.ch09.Ch09_06*.

Débogage de projets pour le Web

Le plug-in Tomcat permet aussi de déboguer interactivement du code de servlet. Imaginons par exemple que l'on place un point d'arrêt dans le code du servlet Ch09_06 que l'on vient juste de développer sur la ligne suivante :

```
out.println("<HTML>");
out.println("<HEAD>");
out.println("<TITLE>");
out.println("une page web");
out.println("</TITLE>");
```

```
out.println("</HEAD>");
out.println("<BODY>");
out.println("<H1>");
out.println("Utilisation de servlets");
out.println("</H1>");
out.println("Utilisation du plug-in Tomcat...");
out.println("</BODY>");
out.println("</HTML>");
```

Lorsque Tomcat est lancé et lorsqu'on navigue sur ce servlet au moyen d'un explorateur, l'exécution va s'arrêter sur le point d'arrêt : le débogueur d'Eclipse va prendre la main, comme on peut le voir dans la figure 9-16. On peut alors exécuter le code pas à pas dans le servlet, comme pour un débogage traditionnel.

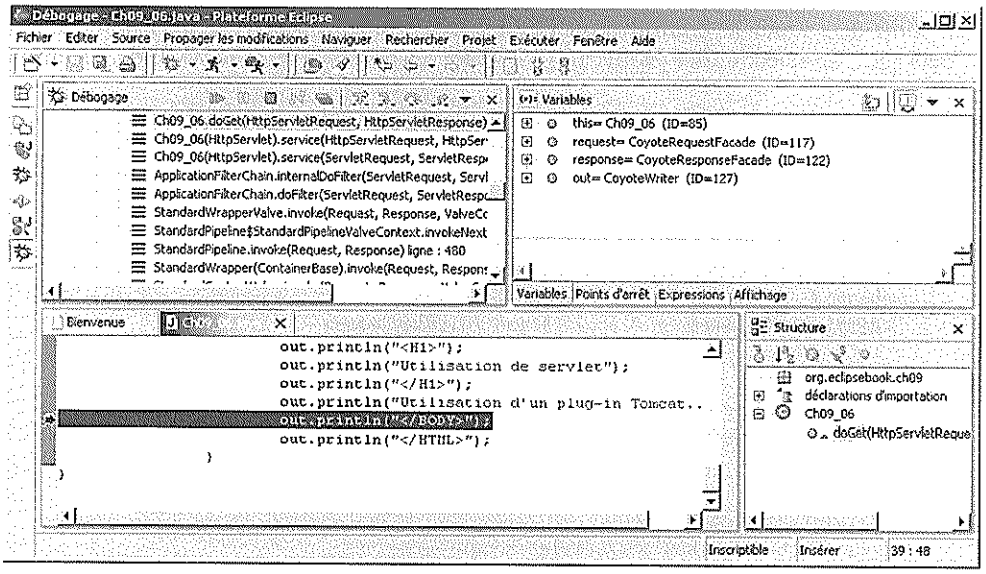


Figure 9-16. Débogage interactif d'un servlet

On peut aussi déboguer des JSP, mais il faut travailler sur le code servlet traduit à partir de la JSP. Par exemple, la JSP *Ch09_04.jsp* du répertoire *webapps/Ch09* est traduite en le code servlet *Ch09_04_jsp.java* (ce nom peut varier en fonction de la version de Tomcat) dans le répertoire *work/localhost/ch09*, avant son exécution (sous les versions les plus récentes de Tomcat, ce répertoire est devenu *work/Standalone/localhost/ch09*). Le fichier servlet est le fichier réellement débogué.

Déploiement d'applications pour le Web

Le dernier sujet examiné dans ce chapitre concerne le déploiement d'applications pour le web. Ce processus est simple – il suffit de créer un fichier compressé au format Web ARchive (WAR) de l'application et de le placer dans le répertoire *webapps*. Au lancement suivant de Tomcat, celui-ci va décompresser automatiquement ce fichier WAR et déployer l'application.

Pour illustrer ce point, nous allons déployer le servlet de l'exemple 9-11. Pour faciliter la création du fichier WAR, on crée un nouveau projet, Ch09_07, dans un sous-répertoire de *webapps* nommé *Ch09_07*. Ce projet est un projet Java standard (et pas un projet Tomcat). Il convient de s'assurer que le répertoire *Ch09_07* possède bien son propre sous-répertoire *WEB-INF* et que celui-ci contient bien *classes* et *lib* ; il faut aussi s'assurer que le répertoire *classes* constitue bien le répertoire de sortie du projet. On peut alors créer *Ch09_07.java* comme précédemment et, après avoir tapé le code du servlet dans *Ch09_07.java*, on peut construire le projet.

Exemple 9-11. Un servlet à déployer

```
package org.eclipsebook.ch09;

import javax.servlet.http.HttpServlet;

import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class Ch09_07 extends HttpServlet {
        public void doGet(HttpServletRequest request,
                HttpServletResponse response)
                throws IOException, ServletException
        {
                response.setContentType("text/html");
                PrintWriter out = response.getWriter();

                out.println("<HTML>");
                out.println("<HEAD>");
                out.println("<TITLE>");
                out.println("Une page web");
                out.println("</TITLE>");
                out.println("</HEAD>");
                out.println("<BODY>");
                out.println("<H1>");
                out.println("Déploiement de projet");
                out.println("</H1>");
                out.println("</BODY>");
                out.println("</HTML>");
        }
}
```

Il faut aussi ajouter le fichier *web.xml* de l'exemple 9-12 au répertoire *WEB-INF* du projet pour faire connaître à Tomcat l'existence de ce servlet. Celui-ci fonctionne alors, comme on peut le vérifier en navigant sur *http://localhost:8080/Ch09_07/org.eclipsebook.ch09.Ch09_07*.

Exemple 9-12. Fichier web.xml pour le servlet à déployer

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE web-app PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application
                          2.3//EN"
"http://java.sun.com/dtd/web-app_2_3.dtd">
<web-app>
```

Exemple 9-12. Fichier web.xml pour le servlet à déployer (suite)

```
    <display-name>Exemple d'application</display-name>

    <servlet>
      <servlet-name>Ch09_07</servlet-name>
      <servlet-class>org.eclipsebook.ch09.Ch09_07</servlet-class>
    </servlet>

    <servlet-mapping>
      <servlet-name>Ch09_07</servlet-name>
      <url-pattern>/org.eclipsebook.ch09.Ch09_07</url-pattern>
    </servlet-mapping>

</web-app>
```

L'étape suivante consiste à déployer ce servlet en le compressant en un fichier WAR. Ceci peut être fait très facilement sous Eclipse au moyen de Ant, en s'appuyant sur le fichier *build.xml* de l'exemple 9-13, qui va compresser tous les fichiers du projet en un fichier WAR nommé *Ch09_07war.war*.

Exemple 9-13. Fichier build.xml pour le servlet à déployer

```
<?xml version="1.0" encoding = "UTF-8"?>
<project name="Ch09_07" default="Main Build" basedir=".">

        <property name="bin"
        location="C:/Program Files/jakarta-tomcat-4.1.30/webapps/ch09_07"/>
        <property name="wardir"
        location="C:/Program Files/jakarta-tomcat-4.1.30/webapps/ch09_07"/>
        <property name="warfile" location="${wardir}/Ch09_07war.war"/>
         <property name="build.compiler"
        value="org.eclipse.jdt.core.JDTCompilerAdapter"/>

        <target name="Main Build" depends="War">
                <echo message="Ant au travail"/>
        </target>

        <target name="War" >
                <jar destfile="${warfile}" basedir="${bin}"/>
        </target>

</project>
```

Pour créer le fichier WAR, il suffit d'ajouter *build.xml* au projet, puis de cliquer le bouton droit de la souris sur *build.xml*, de sélectionner Exécuter Ant et de cliquer le bouton Exécuter pour créer le fichier WAR *Ch09_07war.war*.

On peut déployer l'ensemble de l'application web avec ce seul fichier – il suffit de le placer dans le répertoire Tomcat *webapps* de l'utilisateur de la plateforme et de relancer Tomcat. S'inspirant du nom du fichier WAR, Tomcat va installer l'application pour le web dans un répertoire nommé *Ch09_07war* (on notera que l'on n'a pas nommé ce fichier *Ch09_07.war*, car lorsque Tomcat décompresse le fichier WAR, il aurait écrasé le répertoire *Ch09_07* existant).

Une fois l'application déployée et installée par Tomcat de cette façon, on peut naviguer sur *http://localhost:8080/Ch09_07war/org.eclipsebook.ch09.Ch09_07* à l'aide d'un explorateur (voir figure 9-17). Le servlet a été complètement déployé.

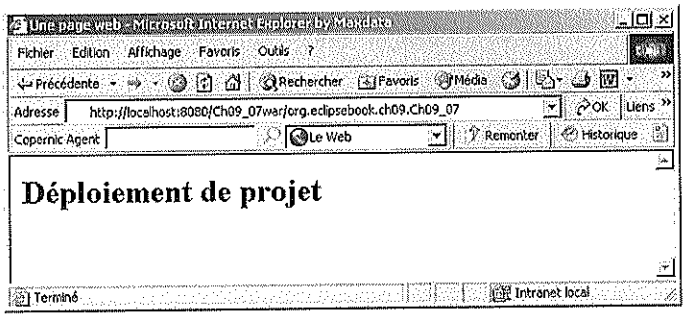


Figure 9-17. Déploiement d'un projet

Ainsi s'achève le chapitre sur Eclipse et le développement d'applications pour le Web. Comme on a pu le voir, l'utilisation d'Eclipse dans ce contexte est très naturelle. La création de JSP ou de servlets, l'installation de fichiers, le lancement ou l'arrêt de Tomcat, ou encore la création de paquetages de déploiement pour le Web, ne posent aucun problème sous Eclipse.

CHAPITRE 10

Développement d'applications Struts avec Eclipse

Ce chapitre est consacré à l'écriture d'applications Struts pour le Web sous Eclipse. Cet exposé permettra au lecteur d'acquérir des compétences non seulement sur l'utilisation de Struts mais aussi sur la création d'applications de grande envergure pour le Web, incluant en particulier des dépendances de génération (i.e. lorsque la compilation d'un fichier doit être effectuée avant celle d'autres fichiers), le non-écrasement de fichiers dans les répertoires de sortie lors d'une construction totale (l'effacement de tous les fichiers du répertoire de sortie constitue ce que l'on appelle un nettoyage [1]), l'organisation des fichiers de code source dans un répertoire après la création du projet, etc. On examinera aussi un plug-in Struts populaire, Easy Struts, qui permet de créer des applications Struts sous Eclipse.

Ce chapitre présente Struts de façon relativement générale, mais une certaine expérience de Struts est requise pour une bonne compréhension des exemples – l'objectif de cet ouvrage est de présenter Eclipse et non Struts.

Struts est construit sur la base de l'architecture Modèle-Vue-Contrôleur (MVC), qui est devenue très populaire chez les développeurs de servlets/JSP. L'architecture originelle de la programmation de servlets/JSP (parfois appelée Modèle 1) était quelque peu ad hoc, utilisant les servlets, les JSP et les beans d'une façon laissée au libre choix du programmeur. Depuis cette époque, les applications pour le web ont été développées à beaucoup plus grande échelle, et l'architecture MVC (parfois appelée Modèle 2) a été adoptée. En programmation MVC, la Vue (souvent une JSP) gère l'interface visuelle avec l'utilisateur, le Modèle (souvent un JavaBean) gère la logique interne de l'application et le Contrôleur (souvent un servlet) se charge de la communication d'ensemble entre la Vue et le Modèle ainsi que de l'acheminement des requêtes de l'utilisateur final vers le code de traitement adapté.

En terminologie Struts, la Vue est construite au moyen de *formulaires* (forms) et le Contrôleur au moyen d'*actions*. Le Modèle est souvent implémenté au moyen d'éléments de type *form beans*.

---

(1) Ou *scrubbing*, ce qu'Eclipse effectue par défaut – le nettoyage d'une application Struts peut provoquer l'effacement inopportun de fichiers requis ultérieurement.

Struts et Eclipse

On va tout d'abord examiner un exemple Struts créé sous Eclipse, en s'appuyant sur les dernières versions de Tomcat (version 4.1.30 au moment de la rédaction de cet ouvrage) et de Struts (version 1.1). Lorsqu'on navigue sur la JSP *http://localhost:8080/Ch10_01/Ch10_01.jsp*, on peut voir un menu contenant un certain nombre de contrôles HTML (voir figure 10-1).

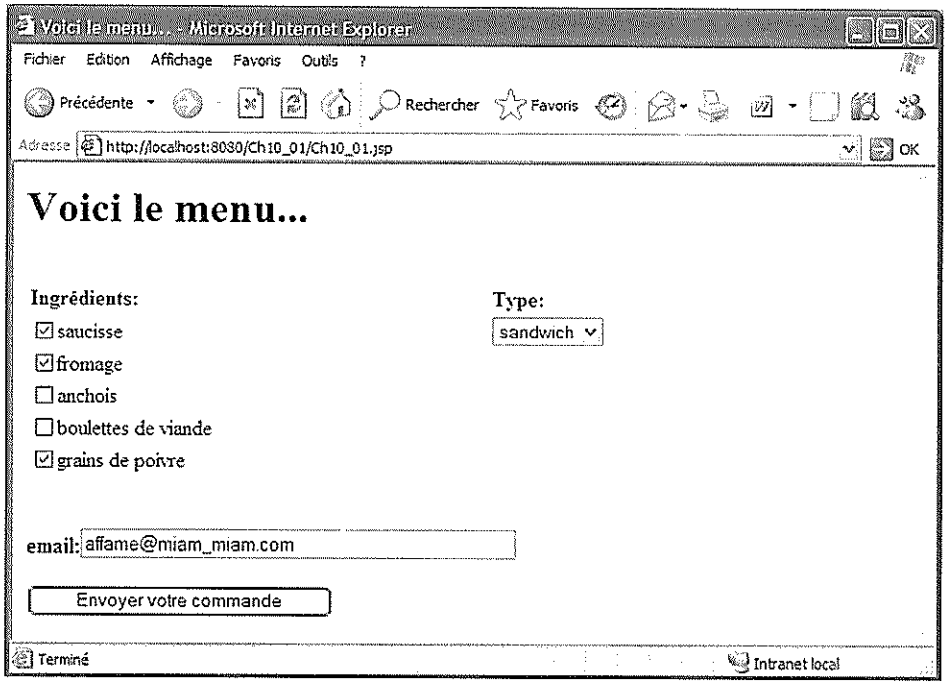


Figure 10-1. La vue Ch10_01.jsp de l'application Struts

L'utilisateur final peut commander une pizza ou un sandwich au moyen des contrôles ; il peut aussi donner son adresse électronique. En cliquant le bouton « Envoyer votre commande », il envoie les données à un bean sous-jacent et au servlet contrôleur de l'application, ce qui provoque l'affichage d'un résumé de la commande (voir figure 10-2).

Bien que tout cela soit en apparence très simple, il n'en est rien lorsqu'on examine l'implémentation de cette application. Struts propose un cadre d'utilisation relativement lourd à mettre en place. Pour créer cet exemple, il est nécessaire d'utiliser les fichiers suivants, stockés dans le répertoire Tomcat *webapps/Ch10_01* :

```
webapps
|
|_ _Ch10_01
   |   Ch10_01.jsp [Vue: formulaire de démarrage]
   |   Ch10_05.jsp [Vue: formulaire de résumé]
   |
   |_ _WEB-INF
      |   struts-config.xml [fichier de configuration Struts]
      |   Ch10_01.tld  [définitions des balises personnalisées]
      |   fichiers TLD Struts [définitions des balises Struts]
      |   web.xml  [fichier descriptif de l'application]
      |
      |_ _lib
      |      struts.jar  [fichier JAR des classes Struts]
      |
      |_ _classes
         |   ApplicationResources.properties [contient les données des
         |   propriétés]
         |
         |_ _org
            |
            |_ _eclipsebook
               |
               |_ _ch10
                  |_ _Ch10_02.class [implémentation de la balise personnalisée 1]
                     Ch10_03.class [implémentation de la balise personnalisée 2]
                     Ch10_04.class [Contrôleur: servlet action]
                     Ch10_06.class [Modèle: élément de type form bean]
```

Il faut maintenant créer cette application sous Eclipse. Pour ce faire, on va créer un projet Java simple, nommé `Ch10_01` ainsi qu'un nouveau sous-répertoire du répertoire Tomcat *webapps*, nommé *Ch10_01*. Pour accéder à ce répertoire sous Eclipse, on crée un nouveau répertoire en cliquant le bouton droit de la souris sur le projet et en sélectionnant Nouveau → Dossier ; on tape alors le nom du répertoire (ici, *deploiement*) et on le lie au répertoire *webapps\Ch10_01*. Sous Eclipse, on crée ensuite les répertoires nécessaires au développement d'applications pour le web – *WEB-INF*, *lib* et *classes* – ce qui déclenchera aussi la création des sous-répertoires de même nom dans le répertoire *webapps\Ch10_01* :

```
deploiement
   |
   |_ _WEB-INF
      |
      |_ _lib
      |
      |_ _classes
```

Pour manipuler le résultat de la compilation de ce projet, il suffit de sélectionner l'option de menu Projet → Propriétés, de cliquer sur le bouton Parcourir se trouvant face à la zone de saisie nommée Dossier de sortie par défaut, de créer un nouveau répertoire, nommé *output*, lié au répertoire *classes*, de s'assurer que celui-ci est bien sélectionné dans la boîte de dialogue Sélection du dossier et de cliquer OK. Tout cela

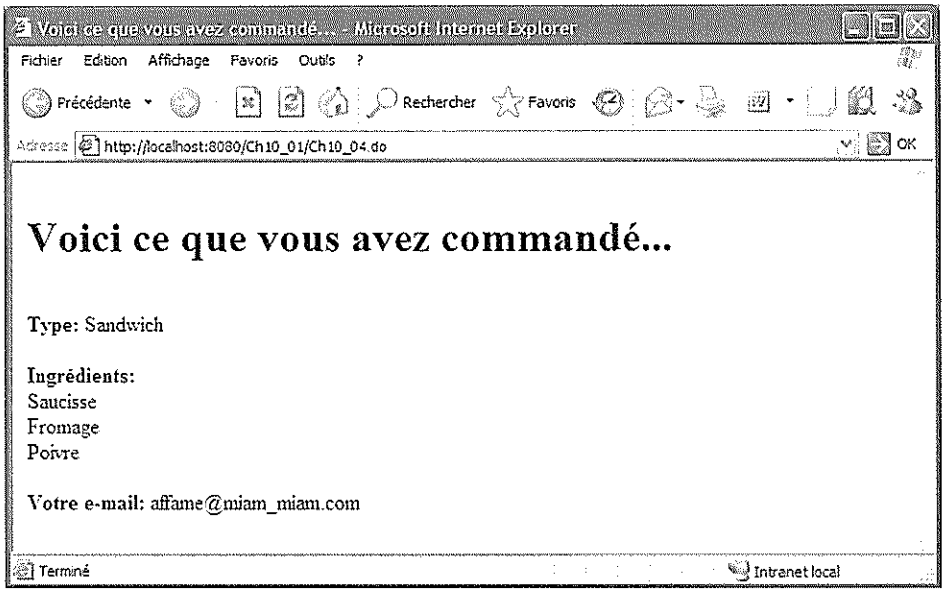


Figure 10-2. Synthèse des fichiers nécessaires à l'application Struts

assure que le code compilé se trouvera bien dans ce répertoire (en réalité, dans le répertoire *classes\org\eclipsebook\ch10*, dans le cas présent).

Dans notre exemple, on n'a délibérément pas créé de répertoire *src* pour stocker le code source à la création du projet (comme on l'a fait dans les chapitres précédents), afin de montrer la simplicité de la création d'un répertoire pour le code source après la création d'un projet. Pour créer un nouveau répertoire de code source, il suffit de cliquer le bouton droit de la souris sur le projet et de sélectionner Nouveau → Dossier source. On donne alors le nom du nouveau répertoire *src*.

Il faut maintenant ajouter *servlet.jar* au chemin de compilation (qui se trouve dans le répertoire Tomcat *common\lib*) ainsi que le fichier JAR contenant Struts, *struts.jar*. Ce fichier est téléchargeable gratuitement sur *http://jakarta.apache.org/struts/*, sous un format compressé (la version courante porte le numéro 1.1). On décompresse ce fichier au moyen de unzip ou de untar pour obtenir *struts.jar*, qui doit être placé dans le répertoire *lib* de l'application. En marge de *struts.jar*, d'autres fichiers JAR sont requis dans cet exemple ; il conviendra de les placer dans ce même répertoire *lib* :

- *commons-beanutils.jar*
- *commons-collections.jar*
- *commons-digester.jar*

Les fichiers Struts *.tld* (Tag Library Definition) suivants doivent aussi être téléchargés et stockés dans le répertoire *WEB-INF* ; ces fichiers permettent d'utiliser les balises personnalisées utilisées par Struts :

- *struts-bean.tld*
- *struts-html.tld*
- *struts-logic.tld*

- *struts-template.tld*

Pour débuter le codage de cet exemple, il faut cliquer le bouton droit de la souris sur le nouveau répertoire et sélectionner Nouveau → Package pour ajouter un nouveau paquetage, *org.eclipsebook.ch10*, au répertoire *src*. Cela initialise l'environnement de génération de code sous Eclipse. On peut enfin commencer à taper du code.

Création de la Vue

Le premier fichier activé par l'utilisateur est le fichier contenant la Vue, *Ch10_01.jsp*. Dans celui-ci, on utilise diverses balises Struts personnalisées pour implémenter l'interface visuelle dans la figure 10-1. Par exemple, la balise `<html:form>` crée un formulaire Struts qui peut afficher des contrôles, comme on peut le voir dans la exemple 10-1 ; on initialise l'attribut `action` du formulaire au nom que l'on va donner au Contrôleur, *Ch10_04.do*, si bien que lorsque l'utilisateur clique le bouton Submit (associé au texte « Envoyer votre commande »), les données du formulaire sont envoyées au Contrôleur.

Exemple 10-1. Un exemple de JSP

```
<%@ page contentType="text/html;charset=UTF-8" language="java" %>
<%@ taglib uri="/WEB-INF/struts-bean.tld" prefix="bean" %>
<%@ taglib uri="/WEB-INF/struts-html.tld" prefix="html" %>
<%@ taglib uri="/WEB-INF/struts-logic.tld" prefix="logic" %>
<%@ taglib uri="/Ch10" prefix="Ch10" %>

<HTML>
    <HEAD>
        <TITLE>Voici le menu...</TITLE>
    </HEAD>

    <BODY>
        <H1>Voici le menu...</H1>
        <html:errors/>
        <Ch10:type/>
        <Ch10:items/>

        <html:form action="Ch10_04.do">
            <TABLE>
                <TR>
                    <TD ALIGN="LEFT" VALIGN="TOP">
                        <bean:message key="items"/>
                        <BR>
                        <logic:iterate id="items1" name="items">
                            <html:multibox property="items">
                                <%= items1 %>
                            </html:multibox>
                            <%= items1 %>
                            <BR>
                        </logic:iterate>
                    </TD>
                    <TD ALIGN="LEFT" VALIGN="TOP">
                        <bean:message key="type"/>
```

Exemple 10-1. Un exemple de JSP (suite)

```
                        <BR>
                        <html:select property="type">
                            <html:options name="type"/>
                        </html:select>
                    </TD>
                </TR>
                <TR>
                    <TD ALIGN="LEFT">
                        <BR>
                        <bean:message key="email"/>
                        <html:text property="email"/>
                    </TD>
                </TR>
            </TABLE>
                <BR>
            <html:submit value="Envoyer votre commande"/>
        </html:form>
    </BODY>
</HTML>
```

On crée ce fichier en cliquant le bouton droit de la souris sur le répertoire *deploiement*, sous Eclipse, et en sélectionnant Nouveau → Fichier. Cela provoque la sauvegarde automatique de *Ch10_01.jsp* dans le répertoire *webapps/Ch10_01*.

Comme on peut le voir dans la figure 10-1, la liste surgissante qui est proposée contient les choix pizza, calzone et sandwich ; on peut aussi voir la liste des ingrédients – saucisse, fromage, anchois, boulettes de viande et grains de poivre – proposés sous la forme de cases à cocher. Pour rendre les choix de ces listes accessibles au contrôle HTML Struts de la Vue, on utilise deux balises JSP personnalisées, `<Ch10:type>` pour retourner des choix tels que pizza et calzone, et `<Ch10:items>` pour retourner des choix tels que saucisse, fromage et anchois. Comme on peut le voir dans l'exemple 10-1, on utilise les balises Struts `<logic:iterate>`, `<html:multibox>` et `<html:options>` pour créer les contrôles HTML requis à partir de ces listes de choix. L'implémentation de ces balises personnalisées figure dans l'exemple 10-2 et dans l'exemple 10-3. Sous Eclipse, on crée donc ces fichiers et on les stocke dans le répertoire *src* en cliquant le bouton droit de la souris sur ce répertoire et en sélectionnant Nouveau → Classe, et en plaçant les nouvelles classes, `Ch10_02` et `Ch10_03`, dans le paquetage `org.eclipsebook.ch10`.

Exemple 10-2. Une classe balise personnalisée, destinée à gérer la commande

```
package org.eclipsebook.ch10;

import javax.servlet.jsp.tagext.TagSupport;

public class Ch10_02 extends TagSupport
{
    public int doStartTag()
      {

        String[] typeArray = {"", "pizza", "calzone", "sandwich"};
```

Exemple 10-2. Une classe balise personnalisée, destinée à gérer la commande (suite)

```
            pageContext.setAttribute("type", typeArray);

            return SKIP_BODY;
        }
    }
```

Exemple 10-3. Une classe balise personnalisée, destinée à gérer la garniture de la pizza

```
package org.eclipsebook.ch10;

import javax.servlet.jsp.tagext.TagSupport;

public class Ch10_03 extends TagSupport
{
    public int doStartTag()
      {
        String[] itemsArray = {"saucisse", "fromage", "anchois",
                               "boulettes de viande", "grains de poivre"};

        pageContext.setAttribute("items", itemsArray);

        return SKIP_BODY;
    }
}
```

Pour que ces balises personnalisées fonctionnent, il est nécessaire de créer un fichier descriptif de la bibliothèque de balises, que l'on peut voir dans l'exemple 10-4. Ce fichier, nommé *Ch10.tld*, doit se trouver dans *deploiement\WEB-INF*.

Exemple 10-4. Le fichier Ch10_01.tld décrivantles balises personnalisées

```
<?xml version="1.0"?>
<!DOCTYPE taglib PUBLIC "-//Sun Microsystems, Inc.//DTD JSP Tag Library 1.1//EN"
    "http://java.sun.com/j2ee/dtds/web-jsptaglibrary_1_1.dtd">

<taglib>
    <tlibversion>1.0</tlibversion>
    <jspversion>1.2</jspversion>
    <shortname>StrutsExample</shortname>
    <info>
        Implémentation de l'exemple Struts
    </info>

    <tag>
        <name>type</name>
        <tagclass>org.eclipsebook.ch10.Ch10_02</tagclass>
        <bodycontent>empty</bodycontent>
    </tag>

    <tag>
        <name>items</name>
        <tagclass>org.eclipsebook.ch10.Ch10_03</tagclass>
        <bodycontent>JSP</bodycontent>
```

Exemple 10-4. Le fichier Ch10_01.tld décrivantles balises personnalisées (suite)

```
    </tag>
</taglib>
```

Création du Contrôleur

Les données contenues dans les contrôles HTML de la Vue peuvent désormais être envoyées à *Ch10_04.do*, qui constitue le Contrôleur (encore appelé servlet action) de notre application. On va maintenant connecter l'extension *.do* au servlet action de l'application dans *web.xml*. On peut voir comment cela est réalisé en examinant le fichier *web.xml* de l'exemple 10-5 ; en particulier, l'utilisation des éléments <servlet> et <servlet-mapping> rappellent sensiblement ce qui a été fait dans le chapitre précédent.

Exemple 10-5. web.xml

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<!DOCTYPE web-app
  PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.2//EN"
  "http://java.sun.com/j2ee/dtds/web-app_2_2.dtd">

<web-app>
        <display-name>exemple d'application Struts</display-name>

  <!-- Action Servlet Configuration -->
  <servlet>
    <servlet-name>action</servlet-name>
    <servlet-class>org.apache.struts.action.ActionServlet</servlet-class>
    <init-param>
      <param-name>application</param-name>
      <param-value>ApplicationResources</param-value>
    </init-param>
    <load-on-startup>2</load-on-startup>
  </servlet>

  <!-- Action Servlet Mapping -->
  <servlet-mapping>
    <servlet-name>action</servlet-name>
    <url-pattern>*.do</url-pattern>
  </servlet-mapping>

  <taglib>
    <taglib-uri>/Ch10_01</taglib-uri>
    <taglib-location>/WEB-INF/Ch10_01.tld</taglib-location>
  </taglib>

  <taglib>
    <taglib-uri>/WEB-INF/struts-bean.tld</taglib-uri>
    <taglib-location>/WEB-INF/struts-bean.tld</taglib-location>
  </taglib>
```

Exemple 10-5. web.xml (suite)

```
    <taglib>
      <taglib-uri>/WEB-INF/struts-html.tld</taglib-uri>
      <taglib-location>/WEB-INF/struts-html.tld</taglib-location>
    </taglib>

    <taglib>
      <taglib-uri>/WEB-INF/struts-logic.tld</taglib-uri>
      <taglib-location>/WEB-INF/struts-logic.tld</taglib-location>
    </taglib>

  </web-app>
```

Ce fichier, *web.xml*, doit se trouver dans le répertoire *deploiement\WEB-INF*, dans le projet Eclipse. L'extension *.do* est ainsi connectée au servlet action de notre application, mais on n'a pas encore spécifié le code proprement dit de ce servlet. Cela est réalisé dans le fichier *struts-config.xml*, qui doit aussi se trouver dans le répertoire *deploiement\WEB-INF*. Pour cet exemple, le code du servlet action sera stocké dans *Ch10_04.class*, alors que la page présentant le résumé de la commande, si tout s'est bien passé, sera stockée dans *Ch10_05.jsp* ; quant au bean qui stockera les données de l'utilisateur, il se trouvera dans *Ch10_06.class*. On peut étudier le fonctionnement de tout ce code dans l'exemple 10-6.

Exemple 10-6. struts-config.xml

```
<?xml version="1.0" encoding="ISO-8859-1" ?>

<!DOCTYPE struts-config PUBLIC
          "-//Apache Software Foundation//DTD Struts Configuration 1.1//EN"
          "http://jakarta.apache.org/struts/dtds/struts-config_1_1.dtd">

<struts-config>

  <form-beans>
    <form-bean name="Ch10_06" type="org.eclipsebook.ch10.Ch10_06"/>
  </form-beans>

  <action-mappings>
    <action path="/Ch10_04"
      type="org.eclipsebook.ch10.Ch10_04"
      name="Ch10_06"
      scope="request"
      input="/Ch10_01.jsp">
      <forward name="OK" path="/Ch10_05.jsp"/>
    </action>

  </action-mappings>

</struts-config>
```

Les données de la Vue peuvent alors être envoyées au bean `Ch10_06`, le Contrôleur `Ch10_04` prenant alors la main. Le Contrôleur reçoit une instance de la classe bean qui contient les données saisies par l'utilisateur dans les contrôles de la vue HTML ; ce Contrôleur vérifie les données, comme on peut le voir dans l'exemple 10-7. Si l'uti-

lisateur n'a pas saisi toutes les informations requises pour la commande, un message d'erreur est affiché ; si les données sont bonnes, elles sont transmises à la page de résumé, *Ch10_05.jsp*. Le fichier contenant le code du Contrôleur, *Ch10_04.java*, doit se trouver dans le répertoire *src*, comme tous les autres fichiers sources Java de ce projet.

Exemple 10-7. La classe de transmission Struts

```
package org.eclipsebook.ch10;

import org.eclipsebook.ch10.Ch10_06;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import org.apache.struts.action.*;

public class Ch10_04 extends Action
{
    public ActionForward execute(ActionMapping mapping,
                ActionForm form,
                HttpServletRequest request,
                HttpServletResponse response)
        throws Exception {

        ActionErrors actionerrors = new ActionErrors();

        Ch10_06 orderForm = (Ch10_06)form;

        String email = orderForm.getEmail();
        if(email.trim().equals("")) {
            actionerrors.add(ActionErrors.GLOBAL_ERROR, new
                ActionError("error.noemail"));
        }

        String type = orderForm.getType();
        if(type.trim().equals("")) {
            actionerrors.add("ActionErrors.GLOBAL_ERROR", new
                ActionError("error.notype"));
        }

        String[] items = orderForm.getItems();
        if(items == null) {
            actionerrors.add("ActionErrors.GLOBAL_ERROR", new
                ActionError("error.noitems"));
        }

        if(actionerrors.size() != 0) {
            saveErrors(request, actionerrors);
            return new ActionForward(mapping.getInput());
        }

        return mapping.findForward("OK");
    }
}
```

Exemple 10-7. La classe de transmission Struts (suite)

Les messages d'erreur référencés dans le code du servlet action sont stockés dans le fichier *ApplicationResources.properties*, qui doit se trouver dans le répertoire *deploiement\WEB-INF\classes*, dans le projet Eclipse. Un listing de ce fichier figure dans l'exemple 10-8 (il est connecté au servlet action dans *web.xml*, voir exemple 10-5).

Exemple 10-8. ApplicationResources.properties

```
email=<b>Your email:</b>
type=<b>Type:</b>
items=<b>Items:</b>

error.noemail=<li><font color="red">Tapez votre adresse électronique
                        SVP.</font></li>
error.notype=<li><font color="red">Sélectionnez un type SVP.</font></li>
error.noitems=<li><font color="red">Sélectionnez au moins un choix
                        SVP.</font></li>

errors.header=<font color="red">Corrigez la ou les erreur(s) suivante(s) SVP
                        :<ul>
errors.footer=</ul>
```

Si l'utilisateur omet certaines données dans la Vue, il verra apparaître l'un des messages d'erreur de *ApplicationResources.properties*, comme on peut le voir dans la figure 10-3.

Création du Modèle

C'en est presque fini du code pour ce projet ; les deux derniers fichiers de code concernent la page de résumé (qui affiche les choix de l'utilisateur), *Ch10_05.jsp*, et le bean jouant le rôle de Modèle et stockant les données fournies par l'utilisateur, *Ch10_06.java*. Dans la page de résumé, *Ch10_05.jsp*, on utilise les données passées par la Vue dans un objet bean et dans diverses balises Struts pour afficher les données (voir exemple 10-9). Ce fichier doit se trouver dans le répertoire *deploiement*, de la même façon que le Vue originelle, *Ch10_01.jsp*.

Exemple 10-9. La JSP destinée à afficher les informaions relatives à la commande

```
<%@ taglib uri="/WEB-INF/struts-bean.tld" prefix="bean" %>
<%@ taglib uri="/WEB-INF/struts-html.tld" prefix="html" %>
<%@ taglib uri="/WEB-INF/struts-logic.tld" prefix="logic" %>

<HTML>
    <HEAD>
        <TITLE>Vous avez effectué la commande suivante...</TITLE>
    </HEAD>

    <BODY>
        <H1>Vous avez effectué la commande suivante...</H1>
        <bean:message key="type"/>
        <bean:write name="Ch10_06" property="type"/>
        <BR>
        <BR>
```

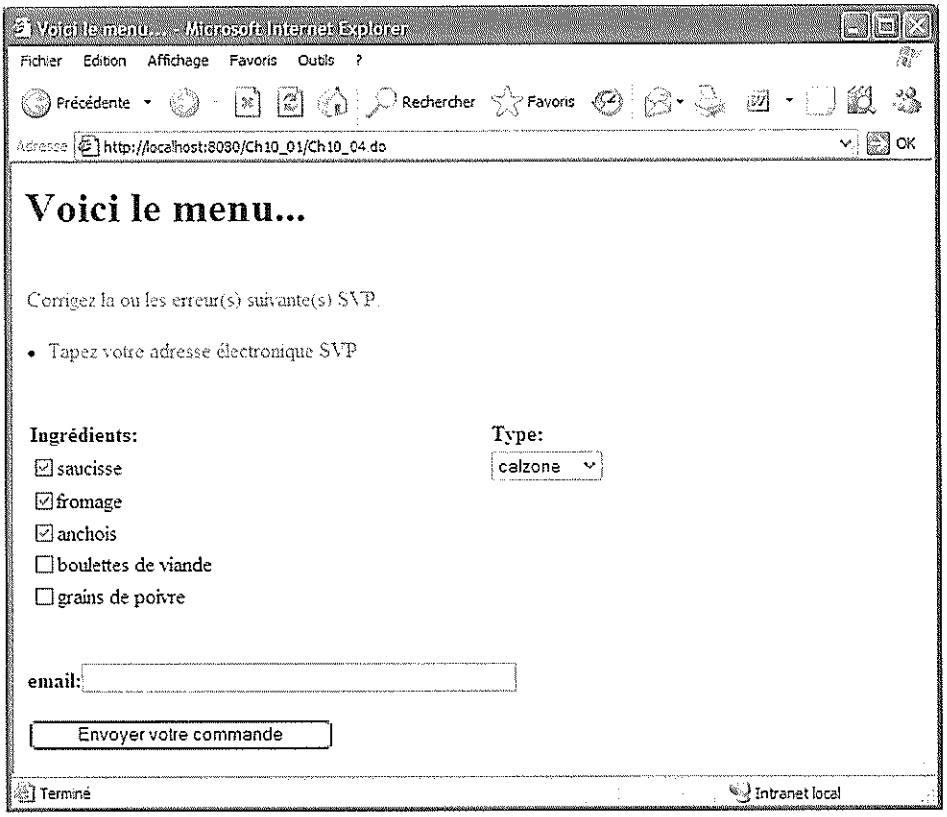


Figure 10-3. Gestion d'une erreur

Exemple 10-9. La JSP destinée à afficher les informaions relatives à la commande (suite)

```
            <bean:message key="items"/>
            <BR>
            <logic:iterate id="items1" name="Ch10_06" property="items">
                <%= items1 %>
                <BR>
            </logic:iterate>
            <BR>
            <bean:message key="email"/>
            <bean:write name="Ch10_06" property="email"/>
            <BR>
        </BODY>
    </HTML>
```

Ch10_06.java, qui contient le bean stockant les données saisies par l'utilisateur, gère les propriétés correspondant à ces données, afin de les rendre accessibles au Contrôleur et à la page de résumé. L'exemple 10-10 présente son fonctionnement. Ce fichier, comme tous les autres fichiers *.java* du projet, doit se trouver dans le répertoire *src*.

Exemple 10-10. Gestionnaire du formulaire

```
package org.eclipsebook.ch10;
import org.apache.struts.action.ActionForm;

public class Ch10_06 extends ActionForm
{

    private String email = "";
    private String type = "";
    private String[] items;

    public String getEmail()
      {
        return email;
    }

    public void setEmail(String email)
      {
        this.email = email;
    }

    public String getType()
      {
        return type;
    }

    public void setType(String type)
      {
        this.type= type;
    }

    public String[] getItems() {
        return items;
    }

    public void setItems(String[] items) {
        this.items = items;
    }
}
```

Ainsi se termine la présentation de l'ensemble du code de l'application – un premier projet de « grande envergure » sous Eclipse – qui constitue le minimum vital requis pour toute application Struts. La structure d'ensemble du projet Eclipse associé à cette application est la suivante :

```
Ch10_01
   |_ _src
   |  |_ _org.eclipsebook.ch10
   |      |_ _ Ch10_02.java
   |      |_ _ Ch10_03.java
   |      |_ _ Ch10_04.java
   |      |_ _ Ch10_06.java
   |
```

```
|_ _JRE System Library
|_ _servlet.jar
|_ _struts.jar
|
|_ _deploiement
   |
   |_ _WEB-INF
   |  |
   |  |_ _classes
   |  |  |_ _ApplicationResources.properties
   |  |
   |  |_ _lib
   |  |  |_ _commons-beanutils.jar
   |  |  |_ _commons-collections.jar
   |  |  |_ _commons-digester.jar
   |  |  |_ _struts.jar
   |  |
   |  |_ _Ch10_01.tld
   |  |_ _struts-bean.tld
   |  |_ _struts-config.xml
   |  |_ _struts-html.tld
   |  |_ _struts-logic.tld
   |  |_ _struts-template.tld
   |  |_ _web.xml
   |
   |_ _Ch10_01.jsp
   |_ _Ch10_05.jsp
```

Avant de procéder à la construction de l'application, il faut noter une dernière chose : le code du servlet action, *Ch10_04.java*, utilise des objets de la classe `Ch10_06` pour stocker les données fournies par l'utilisateur :

```
public class Ch10_04 extends Action
{
    public ActionForward execute(ActionMapping mapping,
                ActionForm form,
                HttpServletRequest request,
                HttpServletResponse response)
        throws Exception {

        ActionErrors actionerrors = new ActionErrors();

        Ch10_06 orderForm = (Ch10_06)form;

        String email = orderForm.getEmail();
        .
        .
        .
```

Cette dépendance pourrait constituer un problème si *Ch10_04.java* était construit avant *Ch10_06.java*. Bien que ce problème puisse survenir avec un outil de construction tel que Ant, cela n'est pas le cas avec Eclipse, qui vérifie les dépendances de génération automatiquement et les résoud en procédant à la compilation des fichiers dans le bon ordre.

D'autre part, si une dépendance circulaire se présente (i.e. si le fichier A dépend du fichier B, qui dépend à son tour de A), une erreur peut survenir. Le programmeur peut activer ou désactiver les dépendances circulaires pour un projet au moyen de la page Fenêtre → Préférences → Java → Compilateur → Chemin de compilation (par défaut, elles sont désactivées), comme on peut le voir dans la figure 10-4. Dans cette boîte de dialogue, on peut demander à Eclipse de traiter les dépendances circulaires comme des erreurs ou des avertissements.

On notera aussi que la case à cocher « Vérifier les dossiers de sortie sur compilation complète » est désélectionnée (voir figure 10-4). Pour apprécier l'importance de la chose, il suffit de constuire le projet en utilisant l'une des options Projet → Recompiler un projet et Projet → Recompiler tout. Celles-ci, qui effectuent la reconstruction totale du projet, donnent le résultat attendu, tous les fichiers *.class* apparaissant dans le sous-répertoire *org\eclipsebook\ch10* du répertoire *classes* :

```
deploiement
   |
   |_ _WEB-INF
      |
      |_ _classes
         |_ _org
            |_ _eclipsebook
               |_ _ch10
                  |_ _Ch10_02.class
                  |_ _Ch10_03.class
                  |_ _Ch10_04.class
                  |_ _Ch10_06.class
```

Cependant, il y a un problème : par défaut, Eclipse procède à un nettoyage des fichiers (*scrubbing*) c'est-à-dire qu'Eclipse efface le contenu du répertoire de sortie, qui a été lié au répertoire *classes*. Ce répertoire contenant le fichier *ApplicationResources.properties*, celui-ci est donc effacé.

Un scrubbing accidentel de ce type est un problème non rencontré lors de la création de plus petits projets, mais il est beaucoup plus fréquent pour des projets comportant un grand nombre de fichiers. Pour résoudre cette question, il faut sélectionner la case à cocher « Vérifier les dossiers de sortie sur compilation complète » de la figure 10-4. Si le projet est reconstruit, *ApplicationResources.properties* sera préservé et les fichiers *.class* seront construits :

```
deploiement
   |
   |_ _WEB-INF
      |
      |_ _classes
         |_ _ApplicationResources.properties
         |_ _org
            |_ _eclipsebook
               |_ _ch10
                  |_ _Ch10_02.class
                  |_ _Ch10_03.class
                  |_ _Ch10_04.class
                  |_ _Ch10_06.class
```

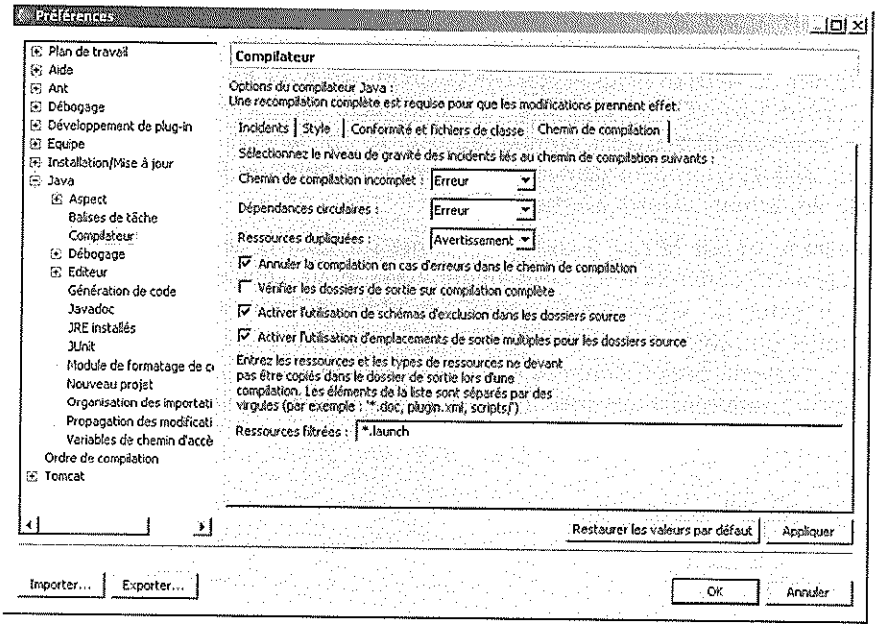


Figure 10-4. Initialisation des options de compilation

Maintenant que le projet est construit, on peut l'exécuter (voir figure 10-1 et figure 10-2).

Comme on a pu le constater, il a fallu jongler avec les fichiers et les répertoires (c'est souvent le cas lors de la création d'applications Struts). Il existe cependant un plug-in Eclipse qui peut faciliter la vie du programmeur : il s'agit du plug-in Easy Struts, que nous allons étudier maintenant.

Utilisation du plug-in Easy Struts

Le plug-in Easy Struts est disponible gratuitement sur *http://sourceforge.net/projects/easystruts*. Ce plug-in ajoute le support Struts à un projet : il propose un certain nombre d'assistants de génération de code, qui permettent de créer des actions, des formulaires, etc.

Pour utiliser ce plug-in, Easy Struts recommande au programmeur de créer tout d'abord un projet Tomcat au moyen du plug-in Tomcat Sysdeo présenté au chapitre 9 (on utilisera donc une version ancienne de Tomcat dans cet exemple, comme on l'a fait dans le chapitre précédent pour le plug-in Tomcat). On nommera ce projet `Ch10_02`.

Pour travailler avec Easy Struts, il faut tout d'abord configurer le plug-in. Pour ce faire, on sélectionne Fenêtre → Préférences, puis l'option Easy Struts et l'onglet Struts 1.1. On ajoute les fichiers JAR et les fichiers TLD Struts, afin que le plug-in sache où les trouver (voir figure 10-5). On laisse inchangées les autres options par défaut et l'on clique OK.

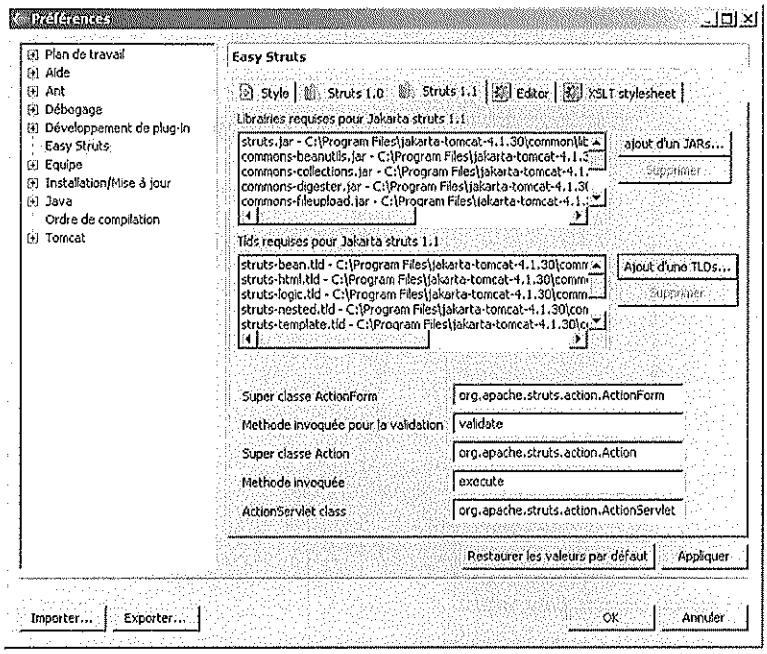


Figure 10-5. Configuration du plug-in Easy Struts

Pour ajouter le support Struts au projet `Ch10_02`, on clique le bouton droit de la souris sur celui-ci et l'on sélectionne Nouveau → Autre, puis l'option Easy Struts dans la liste de gauche ainsi que Add Easy Struts support dans la liste de droite (voir figure 10-6).

On clique Suivant, ce qui provoque l'affichage de la page présentée dans la figure figure 10-7, où l'on peut sélectionner les options Struts ; on définit le paquetage comme étant `org.eclipsebook.ch10` et l'on clique Fin.

Avec la version courante du plug-in, il faut aussi sélectionner Projet → Propriétés, cliquer l'option Chemin de compilation Java, puis l'onglet Ordre et exportation, puis l'élément `TOMCAT_HOME` et le déplacer au moyen du bouton Haut, au-dessus des éléments `TOMCAT_HOME/common/lib/servlet.jar` et `TOMCAT_HOME/lib/jasper-runtime.jar`, afin que `TOMCAT_HOME` soit défini avant d'être utilisé. Cette modification provoque l'effacement du symbole d'avertissement affiché par Eclipse dans l'icône du projet de la Vue Packages.

L'ajout du support Easy Struts provoque l'insertion automatique des fichiers JAR Struts dans le projet. Pour créer un formulaire de saisie ainsi que le bean asssocié, destiné à gérer les données saisies dans ce formulaire, on clique le bouton droit de la souris sur le projet et l'on sélectionne Nouveau → Autre, puis le choix Easy Struts dans la liste de gauche ainsi que le choix Easy Form dans la liste de droite (comme on peut le voir dans la figure 10-6) ; on clique alors Suivant pour ouvrir la boîte de dialogue de la figure 10-8.

On nomme ce nouveau formulaire `Ch10_06`, en accord avec la dénomination du bean créé dans le projet, et on coche la case (Create JSP) input afin qu'Easy Struts crée un

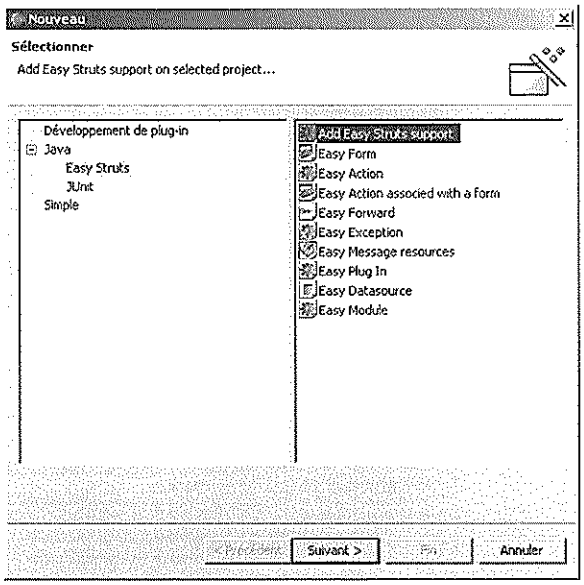


Figure 10-6. Ajout du support Easy Struts

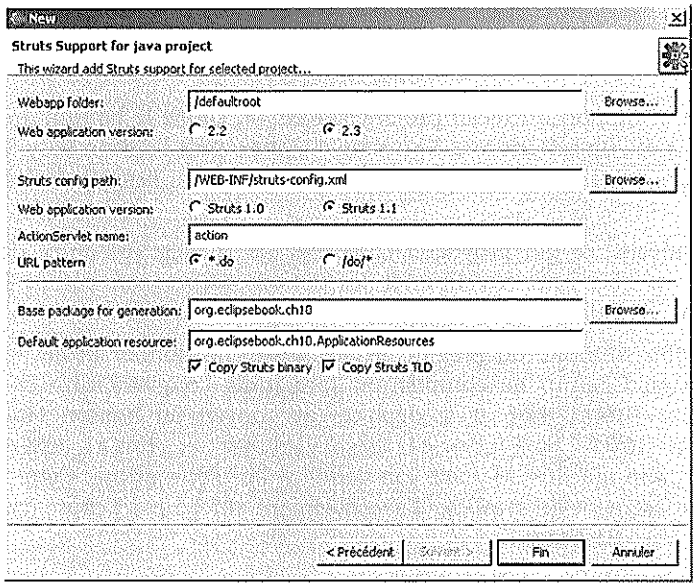


Figure 10-7. Configuration du support Struts

formulaire JSP associé à ce formulaire bean ; on nomme ce formulaire JSP `Ch10_01.jsp`, comme on l'a fait dans un projet précédent.

Easy Struts fournit des outils rudimentaires permettant de créer des propriétés dans le formulaire bean, ainsi que les contrôles HTML associés dans la page JSP. Pour ce faire, il suffit de cliquer le bouton Ajouter se trouvant à côté de la boîte intitulée

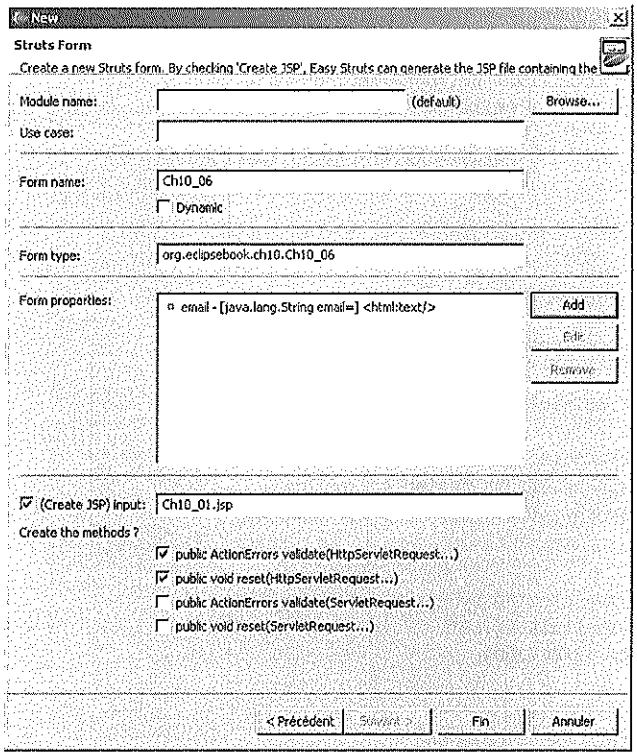


Figure 10-8. Création d'un formulaire

Form properties. Ici, le seul contrôle simple est la boîte de texte éditable permettant à l'utilisateur de taper son adresse électronique (les autres contrôles, une liste de cases à cocher et un contrôle de sélection, ne peuvent être gérés par Easy Struts) : on clique sur le bouton Ajouter, on tape le nom de la propriété à ajouter (qui sera de type `java.lang.String`) et on sélectionne le contrôle de texte HTML destiné à devenir le contrôle associé pour créer le formulaire JSP. On peut alors cliquer Fin pour ajouter la nouvelle propriété au formulaire, puis Fin pour créer *Ch10_01.jsp* et *Ch10_06.java*. Le résultat figure dans l'exemple 10-11 en ce qui concerne *Ch10_01.jsp* et dans l'exemple 10-12 en ce qui concerne *Ch10_06.java*. Voilà est un bon début, mais il reste encore du travail à faire pour obtenir le code développé dans l'exemple précédent.

Exemple 10-11. Génération d'un fichier JSP

```
<%@ taglib uri="/WEB-INF/struts-bean.tld" prefix="bean"%>
<%@ taglib uri="/WEB-INF/struts-html.tld" prefix="html"%>
<html>
    <head>
        <meta name = "Generator" content = "Easy Struts Xslt generator for
                        Eclipse
        (http://easystruts.sf.net)."">

        <title>Struts Form for Ch10_06</title>
     </head>
```

Exemple 10-11. Génération d'un fichier JSP (suite)

```
        <body>
              <html:form action="/[ACTION_PATH]">
                    email : <html:text property="email"/><html:errors
                          property="email"/></br>
                    <html:submit/><html:cancel/>
              </html:form>
        <body>
  </html>
```

Exemple 10-12. Fichier Java généré

```
package org.eclipsebook.ch10;

import javax.servlet.http.HttpServletRequest;

import org.apache.struts.action.ActionErrors;
import org.apache.struts.action.ActionForm;
import org.apache.struts.action.ActionMapping;

/**
 * Ch10_06.java created by EasyStruts - XsltGen.
 * http://easystruts.sf.net
 * created on 12-02-2003
 *
 * XDoclet definition:
 * @struts:form name="Ch10_06"
 */
public class Ch10_06 extends ActionForm {

     // ------------------------------------------------------- Instance
                          Variables

     /** email property */
     private String email;

     // ------------------------------------------------------- Methods

     /**
      * Method validate
      * param ActionMapping mapping
      * @param HttpServletRequest request
      * @return ActionErrors
      */
      public ActionErrors validate(
           ActionMapping mapping,
           HttpServletRequest request) {

           throw new UnsupportedOperationException(
               "Generated method 'validate(...)' not implemented.");
        }

        /**
```

Exemple 10-12. Fichier Java généré (suite)

```
     * Returns the email.
     * @return String
     */
    public String getEmail() {
            return email;
    }

    /**
     * Set the email.
     * @param email The email to set
     */
    public void setEmail(String email) {
            this.email = email;
    }

}
```

Pour créer un servlet action, il suffit de cliquer le bouton droit de la souris sur le projet, de sélectionner Nouveau → Autre puis Easy Struts dans la liste de gauche, comme précédemment, et enfin Easy Action dans la liste de droite, puis de cliquer Suivant. Le boîte de dialogue de la figure 10-9 apparaît alors.

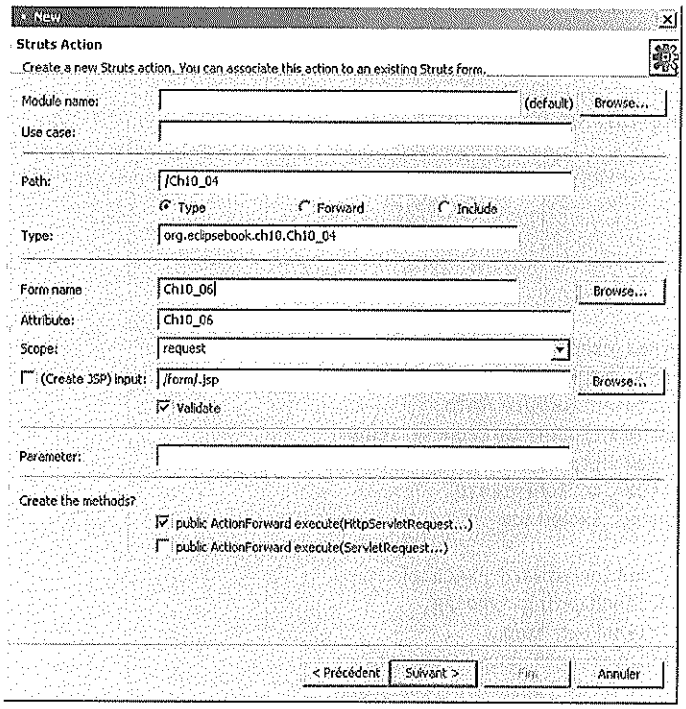


Figure 10-9. Création d'une action

Cliquant sur le bouton Suivant, on accède à la page suivante, dans laquelle on peut créer des liens de type forward vers d'autres servlets ou d'autres JSP ; dans le cas pré-

sent, il suffit de cliquer Fin pour générer le squelette du servlet action *Ch10_04.java* (voir exemple 10-13). Comme on peut le constater, l'essence du servlet action est bien là, mais il lui manque encore beaucoup de code pour obtenir la version précédente de ce fichier.

Exemple 10-13. Génération du code du servlet

```
package org.eclipsebook.ch10;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import org.apache.struts.action.Action;
import org.apache.struts.action.ActionForm;
import org.apache.struts.action.ActionForward;
import org.apache.struts.action.ActionMapping;

/**
 * Ch10_04.java created by EasyStruts - XsltGen.
 * http://easystruts.sf.net
 * created on 12-02-2003
 *
 * XDoclet definition:
 * @struts:action path="/Ch10_04" name="Ch10_06" input="/form/.jsp"
                         validate="true"
 */
public class Ch10_04 extends Action {

    // ------------------------------------------------------- Instance
                         Variables

    // ---------------------------------------------------------- Methods

    /**
     * Method execute
     * @param ActionMapping mapping
     * @param ActionForm form
     * @param HttpServletRequest request
     * @param HttpServletResponse response
     * @return ActionForward
     * @throws Exception
     */
    public ActionForward execute(
        ActionMapping mapping,
        ActionForm form,
        HttpServletRequest request,
        HttpServletResponse response)
        throws Exception {
        Ch10_06 Ch10_06 = (Ch10_06) form;
        throw new UnsupportedOperationException(
            "Generated method 'execute(...)' not implemented.");
```

Exemple 10-13. Génération du code du servlet (suite)

```
        }

    }
```

Commme on peut le constater, Easy Struts fournit la structure élémentaire du code des différents composants d'un programme Struts, mais il revient au programmeur de compléter celle-ci pour obtenir le code final. On ne procèdera pas ici à ce travail ; le lecteur intéressé pourra télécharger le fichier final, `Ch10_02`, sur le site de cet ouvrage – le même que celui créé précédemment dans ce chapitre, mais basé sur Easy Struts.

Ainsi se termine la présentation sur la création manuelle de projets Struts sous Eclipse au moyen d'un plug-in. Les applications Struts requièrent beaucoup de code et de nombreux fichiers, mais cela nous aura permis de développer des applications d'un volume relativement conséquent sous Eclipse. Dans les deux chapitres suivants, nous allons montrer comment développer ses propres plug-in.

CHAPITRE 11

Réalisation d'un plug-in : environnement de développement, fichiers manifestes et points d'extension

L'objectif de ce chapitre et du chapitre suivant est de montrer comment il est possible de modifier l'environnement de développement Eclipse lui-même. Jusqu'à présent, Eclipse a été utilisé tel quel, en ajoutant au besoin quelques plug-in prêts à l'emploi. Nous allons maintenant créer nos propres plug-in. Eclipse a été conçu comme un EDI extensible, possibilité que nous allons exploiter maintenant.

Les plug-in sont stockés dans les sous-répertoires du répertoire *plugins* : on va donc créer nos propres plug-in dans un tel sous-répertoire. En général, ceux-ci portent un nom tel que *org.eclipse.swt.win32_2.1.1* ou *org.junit_3.8.1*, qui permet d'identifier le nom du plug-in et son numéro de version (ces deux informations sont en général séparées par le caractère souligné dans le nom du fichier). Lorsque Eclipse charge un plug-in stocké dans plusieurs sous-répertoires de *plugins*, le numéro de version est utilisé par Eclipse pour que seule la version la plus récente soit chargée.

Tout sous-répertoire de plug-in contient typiquement les fichiers suivants :

**.jar*
: Ces fichiers contiennent le code Java du plug-in

about.html
: Ce fichier est affiché lorsque l'utilisateur souhaite obtenir des informations sur le plug-in

plugin.properties
: Ce fichier contient les chaînes de caractères utilisées par *plugin.xml*

plugin.xml
: Ce fichier est le fichier manifeste décrivant le plug-in pour Eclipse

lib
: Ce répertoire contient des fichiers JAR complémentaires

icons
: Ce répertoire contient des icônes (au format GIF par défaut)

Nous découvrirons ces divers fichiers au fur et à mesure du processus de développement de plug-in décrit dans ce chapitre, ainsi que dans la chapitre suivant. Nous

allons débuter par le seul fichier véritablement indispensable à la création d'un plug-in sous Eclipse : *plugin.xml*. Ce fichier manifeste contient la description du plug-in sous une forme utilisable par Eclipse.

Le fichier manifeste plugin.xml

La création d'un plug-in élémentaire est un jeu d'enfant – il suffit de créer un fichier *plugin.xml* au moyen d'un éditeur de texte. C'est ce que l'on va faire maintenant, dans le but de créer un plug-in fictif nommé `org.eclipsebook.first`. À cette fin, on définit l'id du plug-in, son nom, son numéro de version et le nom de son fournisseur comme suit :

```
<?xml version="1.0" encoding="UTF-8"?>
<plugin
    id="org.eclipsebook.first"
    name="Premier plug-in"
    version="1.0.0"
    provider-name="Steve">
</plugin>
```

Il suffit de stocker *plugin.xml* dans *plugins\org.eclipsebook.first_1.0.0* et de relancer Eclipse. Le nouveau plug-in apparaît alors dans le registre des plug-in, qui est pour Eclipse le lieu de stockage des informations sur tous les plug-in courants. Pour examiner ces informations, il suffit de sélectionner Aide → A propos de Plateforme Eclipse et de cliquer le bouton Détails des plug-in, ce qui fait apparaître la boîte de dialogue A propos des plug-in Plateforme Eclipse de la figure 11-1.

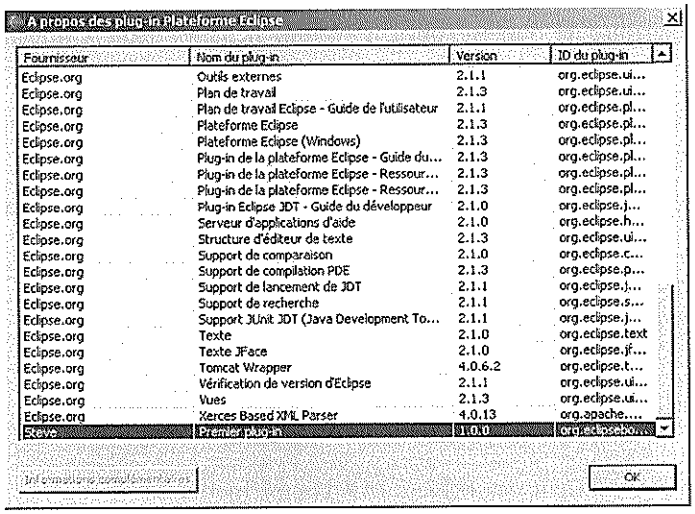


Figure 11-1. Le registre des plug-in

On peut noter la présence du nouveau plug-in fictif en bas du registre.

Si un projet Eclipse comporte un fichier manifeste de plug-in, il est considéré comme un projet de plug-in.

Cet exercice édifiant ne constitue cependant pas une extension éclatante d'Eclipse. La réalisation d'un véritable plug-in implique la création de plusieurs fichiers, et en particulier de fichiers de code. Heureusement, Eclipse propose des assistants de développement de plug-in.

Utilisation de l'environnement de développement de plug-in

La plate-forme Eclipse est par nature un conglomérat de plus d'une centaine de plug-in, construits les uns à partir des autres au moyen de *points d'extension*. Un point d'extension permet de construire un plug-in à partir des exports d'un autre plug-in. Dans ce chapitre, les points d'extension seront utilisés pour ajouter dans Eclipse de nouveaux menus, des boutons, etc., par l'intermédiaire d'un plug-in.

Les plug-in ne peuvent utiliser que les classes exportées par d'autres plug-in, ce qui rend les points d'extension particulièrement importants. Par exemple, si l'on veut qu'un plug-in puisse utiliser du code Java préalablement compilé, il faut attacher des fichiers JAR à celui-ci ou le faire dépendre d'un plug-in contenant ces fichiers JAR. Le support de plug-in personnalisés est déjà implémenté dans de nombreux plug-in fournis en standard avec Eclipse.

L'environnement de développement de plug-in (PDE, Plug-in Development Environment) permet de construire des plug-in en utilisant les points d'extension standard disponibles. Les différents types de projet de plug-in qui peuvent être créés par le PDE sont les suivants :

projet de plug-in
: Le projet de plug-in standard

projet de fragment
: Un add-on ou une addition à un plug-in (parfois utilisé pour l'internationalisation)

projet de dispositif
: Un projet contenant un ou plusieurs plug-in

projet de mise à jour du site
: Un site web pouvant installer automatiquement des dispositifs

Le PDE propose un certain nombre d'assistants prêts à l'emploi, que l'on va utiliser dans le prochain exemple. Celui-ci a pour rôle de créer un plug-in simple supportant un choix de menu et un bouton dans une barre d'outils. Il faut tout d'abord créer le projet de plug-in en sélectionnant l'option de menu Fichier → Nouveau → Projet,

puis le choix Développement de plug-in dans la zone de gauche de la boîte de dialogue Nouveau projet, ainsi que Projet de plug-in dans la zone de droite, comme indiqué dans la figure 11-2. On peut alors cliquer Suivant.

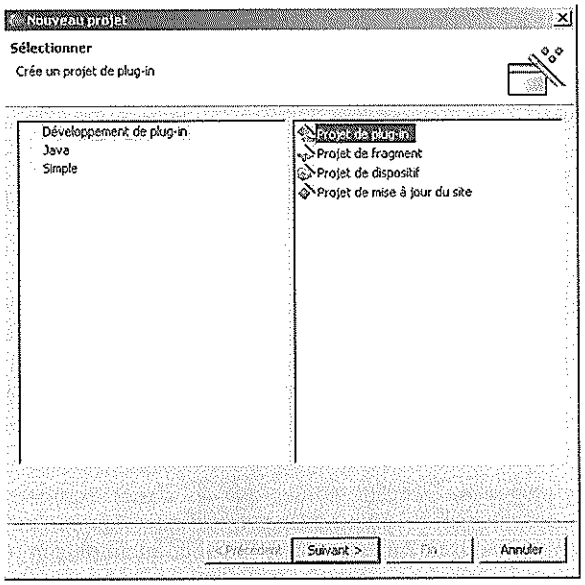


Figure 11-2. Création d'un projet de plug-in

Dans la page suivante, il faut saisir le nom du projet (ici, `org.eclipsebook.ch11.Ch11_01`, voir figure 11-3), avant de cliquer une nouvelle fois Suivant. Le nom du projet sera aussi l'id du plug-in lorsque celui-ci sera utilisé dans Eclipse.

Dans la page suivante, il convient de s'assurer que le bouton radio « Créer un projet Java » est sélectionné, comme on peut le voir dans la figure 11-4, avant de cliquer Suivant.

La page suivante présente les générateurs de code de plug-in (voir figure 11-5). Sélectionnons l'exemple « Hello, World », fourni avec Eclipse, dans la zone de gauche et cliquons encore Suivant (cette page permet aussi de créer d'autres types de projets de plug-in, tels que les plug-in comportant des éditeurs, des vues, etc.).

La page suivante permet de configurer le plug-in. À cette fin, il faut saisir le nom du fournisseur – on utilisera ici `Eclipse Book` –, les autres choix étant ceux par défaut, comme indiqué dans la figure 11-6. On peut alors cliquer Fin pour engendrer le code de ce plug-in (cliquer Suivant à cette étape permet de spécifier le texte affiché par le plug-in dans la boîte de message, mais cela est inutile ici car le texte par défaut nous suffit).

Cliquer Fin provoque l'ouverture de l'éditeur de fichier manifeste de plug-in de la figure 11-7, qui affiche une page de bienvenue. Il est possible d'ouvrir ultérieurement cet éditeur en double-cliquant sur *plugin.xml*, dans la Vue Packages.

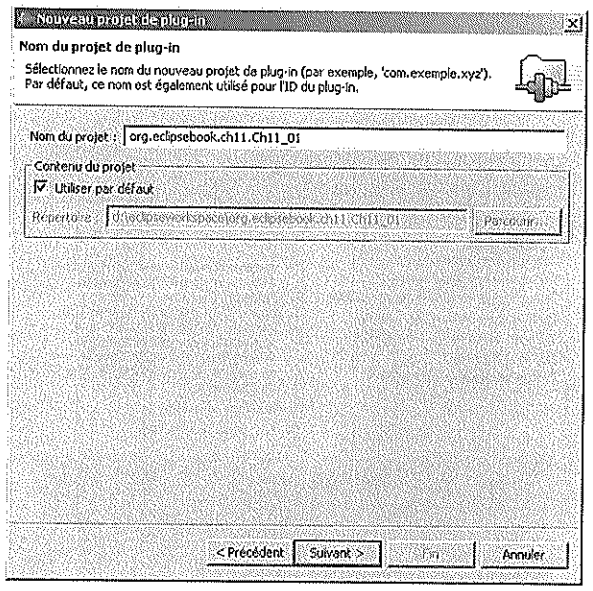


Figure 11-3. Spécification du nom du projet

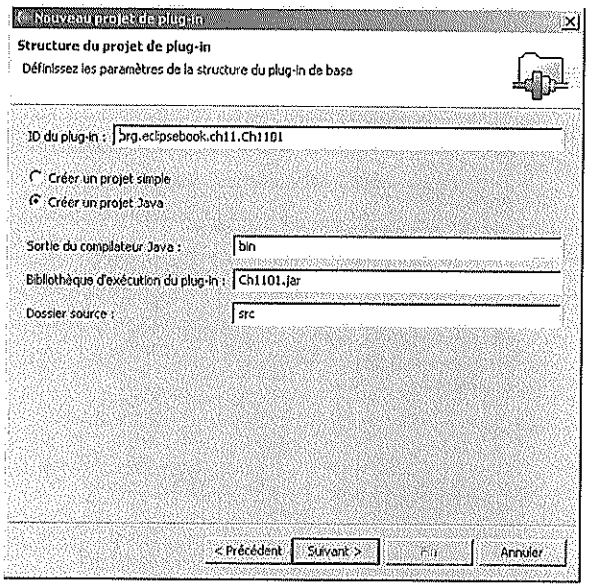


Figure 11-4. Spécification du type de projet

L'éditeur de fichier manifeste de plug-in paraît faussement simple. On notera en particulier les onglets au bas de la fenêtre de l'éditeur :

Bienvenue

La page de bienvenue de la figure 11-7, qui décrit le plug-in

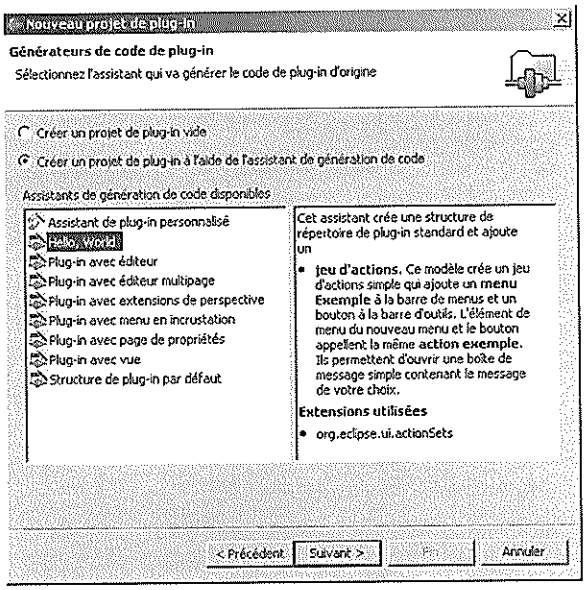


Figure 11-5. Création de l'exemple Hello, World

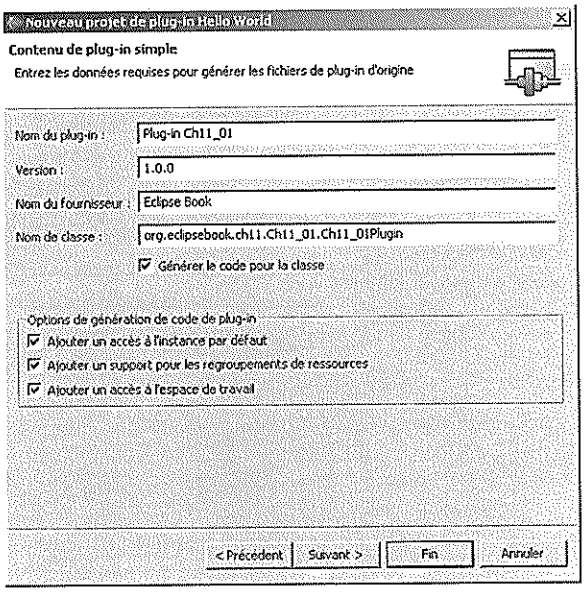


Figure 11-6. Configuration du plug-in

Présentation

Cette page contient une information synthétique sur le plug-in (nom, version, nom du fournisseur, etc.)

Dépendances

Cette page indique les plug-in requis pour ce plug-in

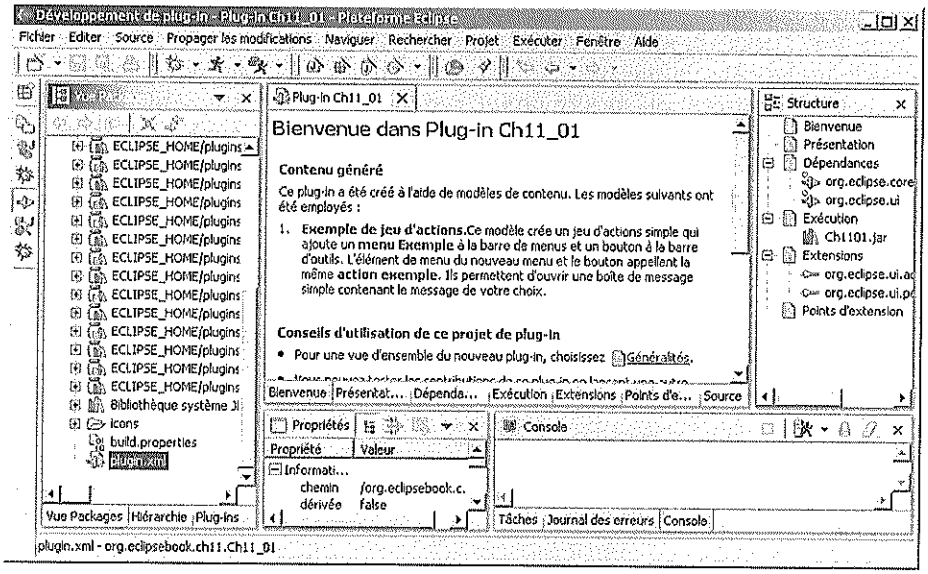


Figure 11-7. L'éditeur de fichier manifeste de plug-in

Exécution
: Cette page indique les bibliothèques requises pour exécuter ce plug-in

Extensions
: Cette page indique les points d'extension utilisés par ce plug-in

Points d'extension
: Cette page indique les points d'extension définis par ce plug-in

Source
: Cette page propose un éditeur XML permettant d'éditer le code source de *plugin.xml*

L'éditeur de fichier manifeste de plug-in permet d'éditer *plugin.xml*, ou bien directement en cliquant l'onglet Source, ou bien au moyen des différents onglets de l'éditeur. Par exemple, pour procéder à une construction via des points d'extension, on peut cliquer l'onglet Extensions et utiliser le bouton Ajouter pour ajouter une fonctionnalité à un plug-in, comme on le fera plus loin dans ce chapitre. Les modifications effectuées de la sorte provoquent une mise à jour automatique de *plugin.xml* par l'éditeur du PDE, sans intervention directe du programmeur sur ce fichier.

Utilisation du plan de travail d'exécution

À cette étape, on pourrait créer le fichier *Ch11_01.jar* et l'installer (ainsi que les autres fichiers requis, dont *plugin.xml*) dans le répertoire *plugins* – le nouveau plug-in supporterait l'option de menu et le bouton. Cependant, Eclipse propose un outil spécifique, permettant d'effectuer des tests : le plan de travail d'exécution. Ce plan de travail, appelable depuis Eclipse, permet de tester les plug-in.

Pour lancer le plan de travail d'exécution, il suffit de sélectionner le projet de plug-in dans la Vue Packages, puis l'option de menu Exécuter → Exécuter ; il faut ensuite ouvrir le nœud plan de travail d'exécution dans la zone de gauche, et enfin l'option plan de travail d'exécution sous ce nœud, comme on peut le voir dans la figure 11-8. On sélectionne ensuite le JRE à utiliser avec le plan de travail d'exécution (cette étape est inutile si le JRE installé a été nommé JRE2, qui est celui cherché par défaut par le plan de travail d'exécution). On peut alors cliquer Exécuter (pour les fois suivantes, il suffira de sélectionner Exécuter → Exécuter en tant que → plan de travail d'exécution).

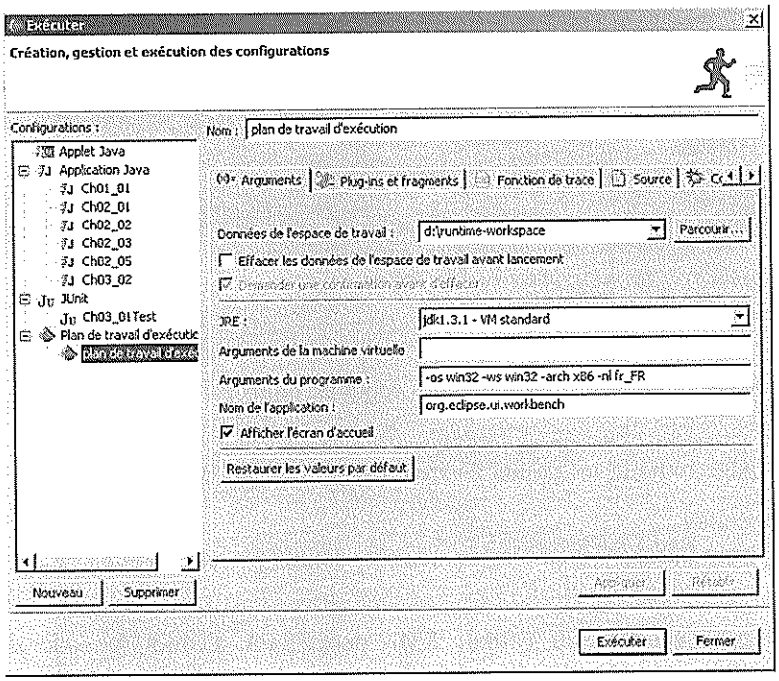


Figure 11-8. Spécification d'une configuration de lancement pour le plan de travail d'exécution

Le plan de travail d'exécution est alors lancé, mais il reste éventuellement encore quelque chose à faire. Si l'action de ce plug-in n'est visible ni dans la barre de menu, ni dans la barre d'outils, il faut sélectionner Fenêtre → Personnaliser la perspective → Autre, puis la case à cocher Exemple de jeu d'actions, qui est définie par notre plug-in, et cliquer OK (si le bouton et l'option de menu sont déjà visibles, c'est que cette case est déjà cochée). Cela provoque l'ajout du menu défini par le plug-in, Exemple de menu, dans la barre de menu, et d'un bouton contenant l'icône Eclipse dans la barre d'outils se trouvant juste au-dessous de ce menu, comme on peut le voir dans la figure 11-9.

Sélectionner l'option Exemple Menu → Exemple d'action ou cliquer le nouveau bouton provoque l'affichage par le plug-in du texte « Hello, Eclipse World » dans une boîte de message (voir figure 11-10).

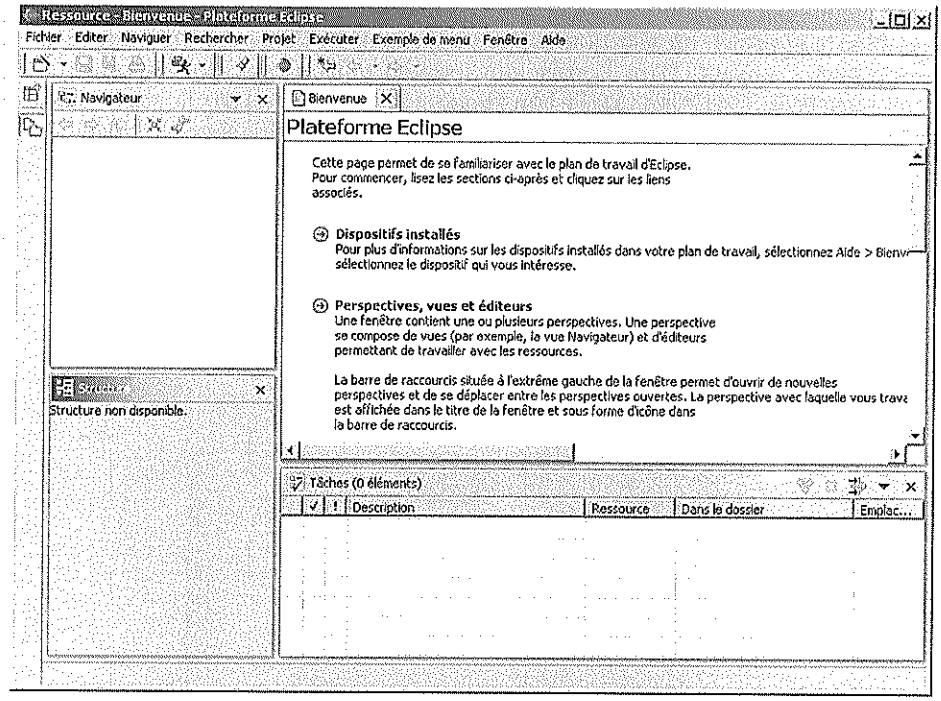


Figure 11-9. Le plan de travail d'exécution

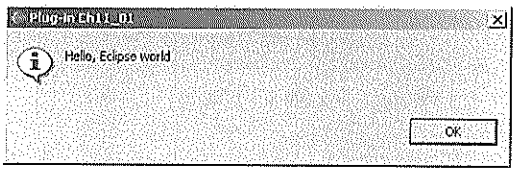


Figure 11-10. La boîte de message du plug-in

Et voilà ! Nous venons de créer un plug-in Eclipse opérationnel. Pour en terminer avec cet exemple, il suffit de fermer le plan de travail d'exécution.

Il est possible de déboguer ce code comme tout autre code sous Eclipse : il suffit pour cela de sélectionner Exécuter → Déboguer en tant que → Plan de travail d'exécution.

Dans le cas présent, les assistants du PDE ont créé le code requis pour ce plug-in simple. Dans l'exemple suivant, nous allons construire le code nous-même.

Création d'un plug-in standard

Dans cet exemple, nous allons prendre en charge tout le processus de création du plug-in qui, comme le précédent, va créer une option de menu et un bouton de barre d'outils.

Commençons par sélectionner Fichier → Nouveau → Projet. Dans la boîte de dialogue Nouveau projet, on sélectionne Développement de plug-in et Projet de plug-in, puis on clique Suivant. Dans la page suivante, on tape le nom du nouveau projet, `org.eclipsebook.ch11.Ch11_02`, et l'on clique Suivant pour ouvrir la page Structure du projet de plug-in. On clique encore Suivant en gardant les choix par défaut.

Dans la page Générateurs de code de plug-in, on sélectionne l'option Structure de plug-in par défaut, comme indiqué dans la figure 11-11, et on clique Suivant.

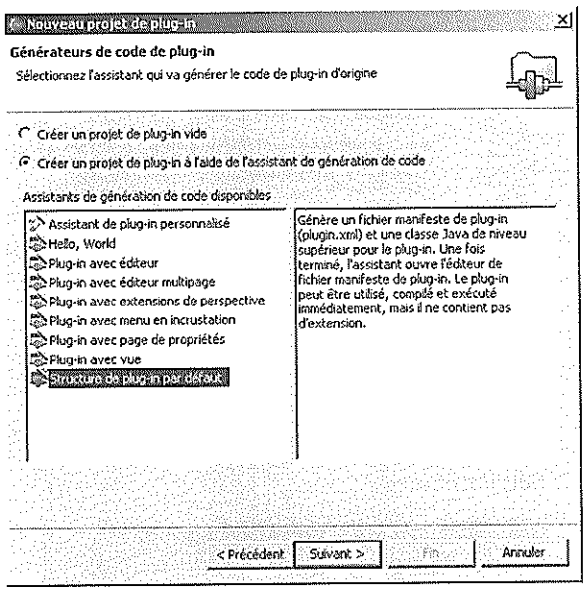


Figure 11-11. Spécification de l'assistant de code du plug-in

Dans la page suivante, Contenu de plug-in simple (voir figure 11-12), on spécifie le nom du fournisseur – `Eclipse Book` dans cet exemple. On n'a pas besoin des méthodes que l'assistant peut générer pour nous, aussi peut-on désélectionner les cases à cocher de la boîte intitulée « options de génération de code de plug-in ». On clique alors le bouton Fin.

Cela devrait générer le plug-in `Ch11_02` et ouvrir l'éditeur de fichier manifeste de plug-in sous Eclipse. En cliquant la page Source, on obtient le code XML de *plugin.xml* :

```
<?xml version="1.0" encoding="UTF-8"?>
<plugin
   id="org.eclipsebook.ch11.Ch1102"
   name="Ch11_02 Plug-in"
   version="1.0.0"
   provider-name="Eclipse Book"
   class="org.eclipsebook.ch11.Ch11_02.Ch11_02Plugin">

   <runtime>
      <library name="Ch1102.jar"/>
   </runtime>
```

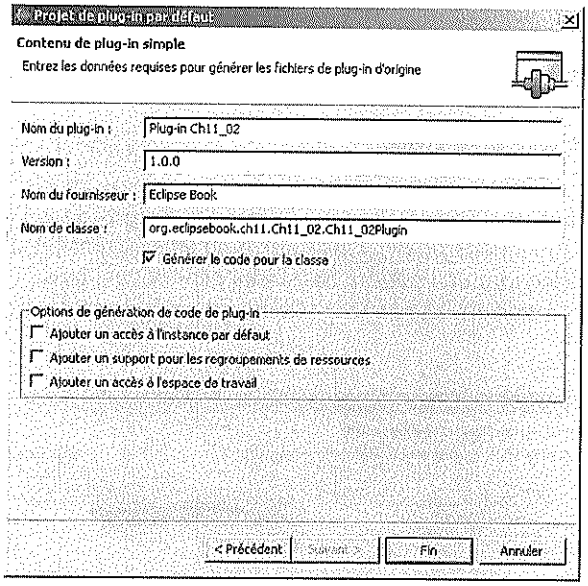


Figure 11-12. Spécification des options de codage

```
    <requires>
       <import plugin="org.eclipse.core.resources"/>
       <import plugin="org.eclipse.ui"/>
    </requires>

</plugin>
```

Ce fichier constitue le fichier manifeste de notre nouveau plug-in, dont le rôle est d'intégrer une nouvelle option de menu ainsi qu'un bouton dans la barre d'outils. Pour ce faire, il faut définir des *actions* et des *jeux d'action* (action sets).

Une action représente une commande que l'utilisateur peut déclencher. Pour implémenter le traitement ainsi déclenché, il faut créer une classe fille de la classe `Action`. Une instance de cette classe peut être utilisée dans des situations très variées, par exemple lorsque l'utilisateur sélectionne une option de menu ou lorsqu'il clique un bouton de la barre d'outils – l'objet de type action sera sollicité dans les deux cas, ce qui déclenchera donc le même traitement. On comprend ainsi comment il est possible de lancer le même traitement à partir d'éléments d'interface graphique différents.

Comme on peut l'imaginer, un jeu d'actions est un ensemble d'actions. On lie des éléments d'interface graphique utilisateur (menus, barres d'outils, etc.) à un ensemble d'actions (l'élément `<actionSet>` dans *plugin.xml* contient typiquement des éléments `<menu>` et `<action>`). Une action est un objet Java qui définit ce qui doit se passer lorsque l'utilisateur agit sur une commande spécifique (option de menu, bouton dans la barre d'outils, etc.) ; un jeu d'actions lie des actions à des éléments d'interface graphique utilisateur.

On crée des jeux d'actions sous la forme d'extensions du point d'extension `org.eclipse.ui.actionSets`, ce qui constitue la manière dont il faut implémenter les

options de menu et les boutons. Il serait possible d'implémenter l'extension sous forme XML directement dans le fichier *plugin.xml*, mais il est bien plus facile d'utiliser l'éditeur de fichier manifeste.

Création d'un jeu d'actions

Pour créer un jeu d'actions permettant de lier un menu et un bouton de barre d'outils à une action, il suffit de cliquer l'onglet Extensions dans l'éditeur de fichier manifeste puis Ajouter pour ouvrir l'assistant Nouvelle Extension de la figure 11-13. Il suffit alors de sélectionner l'option Assistants génériques dans la zone de gauche, ainsi que l'option Extension basée sur le schéma, dans la zone de droite, et de cliquer Suivant.

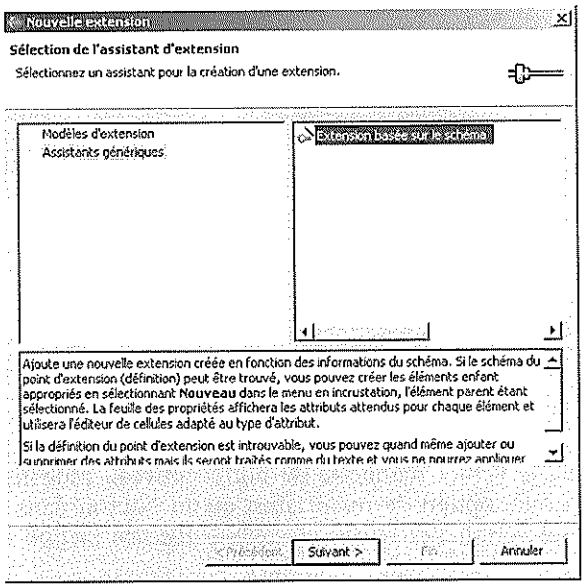


Figure 11-13. L'assistant Nouvelle extension

La page suivante est la page Selection du point d'extension (voir figure 11-14). Dans le cas présent, on souhaite implémenter les traitements déclenchés par l'option de menu et le bouton de la barre d'outils, qui sont tous deux gérés dans notre jeu d'actions : il suffit donc de sélectionner le point d'extension `org.eclipse.ui.actionSets` et de cliquer Fin.

Cette sélection ajoute l'extension de base dont on a besoin pour utiliser la barre d'outils et le menu. Pour créer le jeu d'actions, il faut cliquer le bouton droit de la souris sur le choix `org.eclipse.ui.actionSets` dans la zone intitulée Toutes les extensions et de sélectionner l'option Nouveau → actionSet. Cela permet de créer le nouveau jeu d'actions de la figure 11-15.

Pour définir les propriétés de ce nouveau jeu d'actions, il faut le sélectionner puis, dans la vue Propriétés, initialiser la propriété `label` à New Action Set et la propriété `visible` à true. L'utilisation de l'éditeur de fichier manifeste permet de modifier

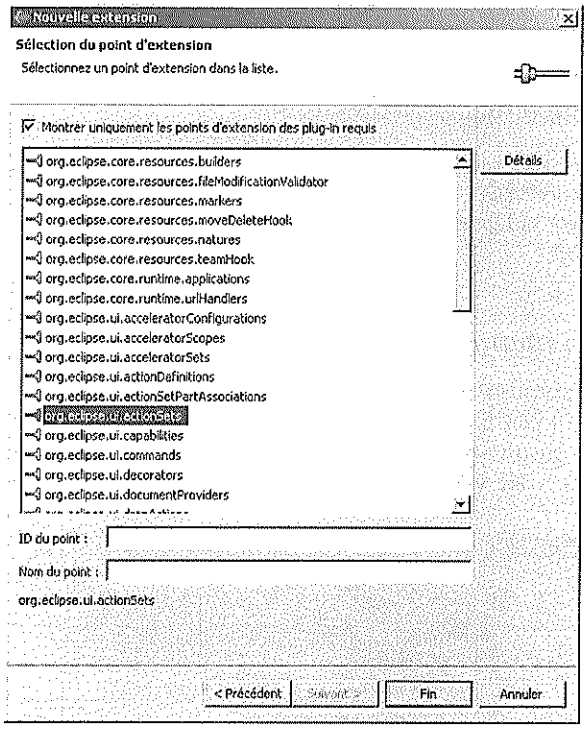


Figure 11-14. Sélection d'un point d'extension

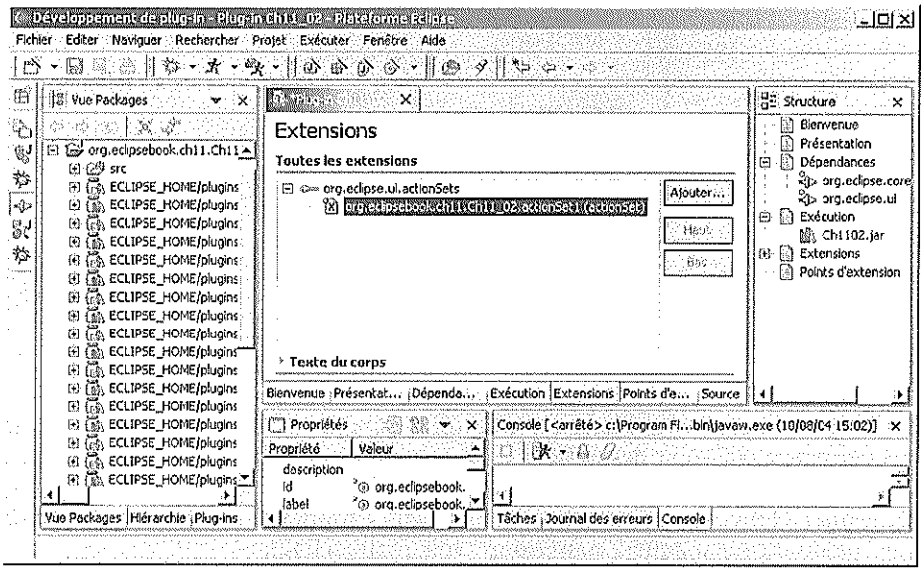


Figure 11-15. Création d'un jeu d'actions

automatiquement le code XML de *plugin.xml*. Voici le code XML ainsi généré ; celui-ci indique que cette extension contient un jeu d'actions :

```
<extension
      point="org.eclipse.ui.actionSets">
   <actionSet
         label="New Action Set"
         visible="true"
         id="org.eclipsebook.ch11.Ch11_02.actionSet1">
   </actionSet>
</extension>
```

Création d'un menu

Pour créer un nouveau menu, il suffit de cliquer le bouton droit de la souris sur New Action Set dans l'onglet Extensions et de sélectionner Nouveau → menu. Dans la vue Propriétés, il faut ensuite initialiser la propriété id du nouveau menu à newMenu, et sa propriété label à New Menu (voir figure 11-16).

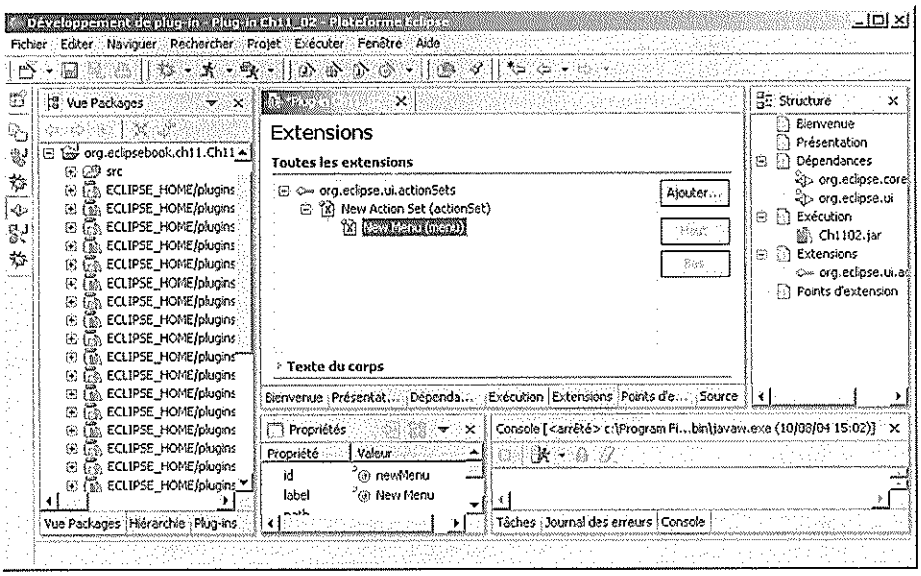


Figure 11-16. Création du nouveau menu

Pour ajouter l'option de menu, on crée tout d'abord un séparateur de menu, qui nous permettra d'ajouter des options dans ce menu lors de l'exécution. Pour ajouter ce séparateur de menu, il faut cliquer le bouton droit de la souris sur New Menu et sélectionner Nouveau → separator. Dans la vue Propriétés, on initialise ici la propriété name du séparateur à newGroup (voir figure 11-17).

Notre jeu d'actions possède ainsi un nouveau menu ; l'étape suivante consiste à lier une action à ce menu. Même s'il y avait plusieurs menus et plusieurs boutons de barre d'outils, il serait possible de les lier tous dans le cadre du même jeu d'actions (ce qui aurait pour effet de grouper ces boutons ensemble dans la barre d'outils).

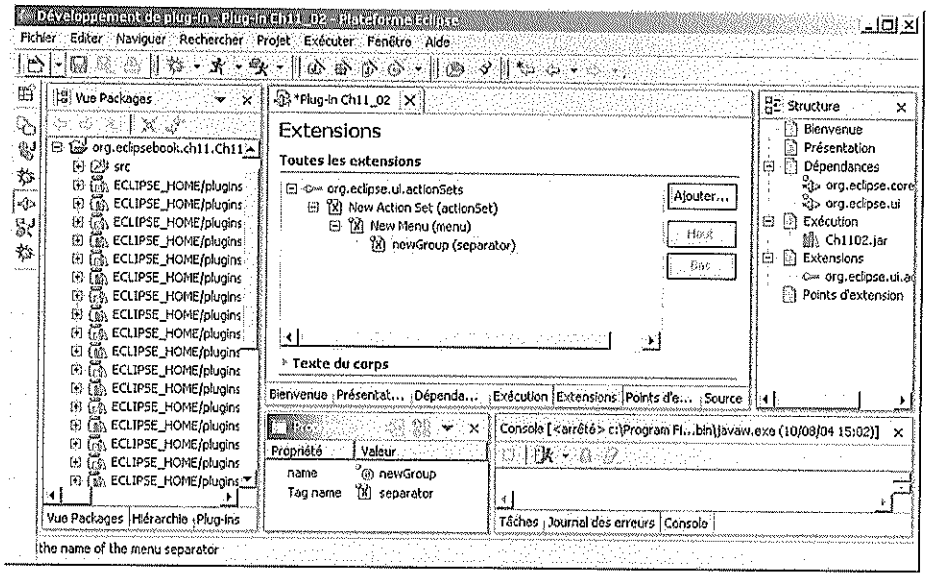


Figure 11-17. Création d'un séparateur de menu

Création d'une action

On ajoute maintenant une nouvelle action pour associer du code Java au jeu d'actions en charge de la gestion du menu et de la barre d'outils. Pour créer une nouvelle action, il faut cliquer le bouton droit de la souris sur l'option New Action Set, dans la boîte Toutes les extensions, et sélectionner Nouveau → action.

Pour personnaliser cette action, on va créer une option de menu et un contrôle de barre d'outils qui vont afficher le message « Bonjour chez vous ». Pour ce faire, il faut sélectionner la nouvelle action dans la zone Extensions. Dans la vue Propriétés, il faut ensuite initialiser la propriété `label` à New Action et la propriété `toolTip` à Bonjour chez vous. Ensuite, il faut initialiser la propriété `menubarPath` à newMenu/newGroup et la propriété `toolbarPath` à newGroup.

Il faut maintenant associer cette action à du code Java. Pour ce faire, il faut sélectionner l'action, New Action, dans la zone Toutes les extensions, et cliquer le bouton comportant des points de suspension (. . .) qui apparaît lorsqu'on sélectionne la propriété `class` dans la vue Propriétés. Cela provoque l'ouverture de l'Editeur d'attribut Java de la figure 11-18. Il faut alors cliquer le bouton radio Générer une nouvelle classe Java, taper `org.eclipsebook.ch11.Ch11_02` dans la boîte intitulée Nom du package (ou cliquer le bouton Parcourir et sélectionner ce paquetage), puis donner un nom à la classe, `NewAction`, comme on peut le voir dans la figure. On clique ensuite sur Fin.

On aboutit ainsi à la création d'un nouvel élément `<action>` dans *plugin.xml* :

```
<action
      label="New Action"
      tooltip="Bonjour chez vous."
      class="org.eclipsebook.ch11.Ch11_02.NewAction"
```

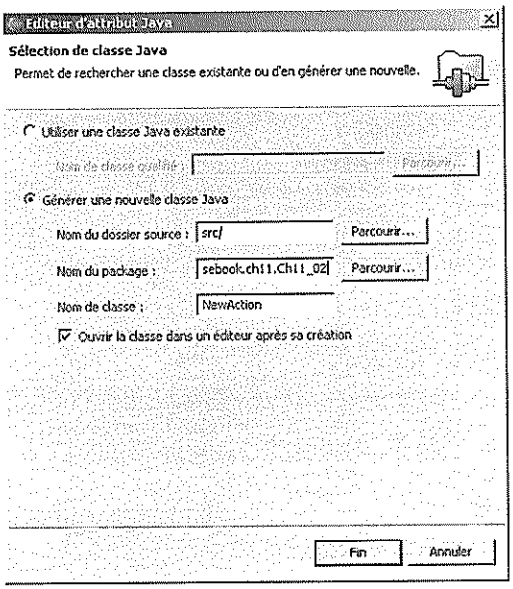


Figure 11-18. Création d'une classe action

```
        menubarPath="newMenu/newGroup"
        toolbarPath="newGroup"
        id="org.eclipsebook.ch11.Ch11_02.action1">
    </action>
```

De plus, un nouveau fichier Java, *NewAction.java*, a été créé et ajouté au projet. Le cadre général du plug-in est désormais en place. Il reste maintenant à écrire le code qui affichera réellement la boîte de message lorsque l'utilisateur cliquera le bouton de la barre d'outils ou l'option de menu.

Écriture du code

On ouvre maintenant *NewAction.java*. Pour afficher une boîte de message, il faut un objet implémentant l'interface `IWorkbenchWindow`. Comme cet objet sera passé en argument de la méthode `init` de *NewAction.java*, il faut commencer par créer une variable de classe private `workbenchWindow`, pour le stocker :

```
public class NewAction implements IWorkbenchWindowActionDelegate {
        private IWorkbenchWindow workbenchWindow;
        .
        .
        .
```

On stockera cet objet dans la méthode `init` de la façon suivante :

```
public void init(IWorkbenchWindow window)  {
        workbenchWindow = window;
}
```

On utilise maintenant dans la méthode run cette variable et la méthode openInformation de la classe MessageDialog pour afficher la boîte de message et son texte « Bonjour chez vous » :

```
public void run(IAction action)  {
        MessageDialog.openInformation(
        workbenchWindow.getShell(),
        "New Plug-in",
        "Bonjour chez vous.");
}
```

Pour utiliser cette classe, il conviendra de procéder à l'importation ad hoc suivante :

```
import org.eclipse.jface.dialogs.MessageDialog;
```

On sauvegarde maintenant tous les fichiers ; le développement du plug-in est achevé. Pour le tester, il faut lancer le plan de travail d'exécution. Pour cela, on sélectionne Fenêtre → Personnaliser la perspective, puis la case à cocher New Action Set. On doit obtenir le résultat de la figure 11-19, où l'on peut voir l'icône de bouton par défaut utilisée pour les boutons de la barre d'outils ainsi que le nouveau menu, New Menu.

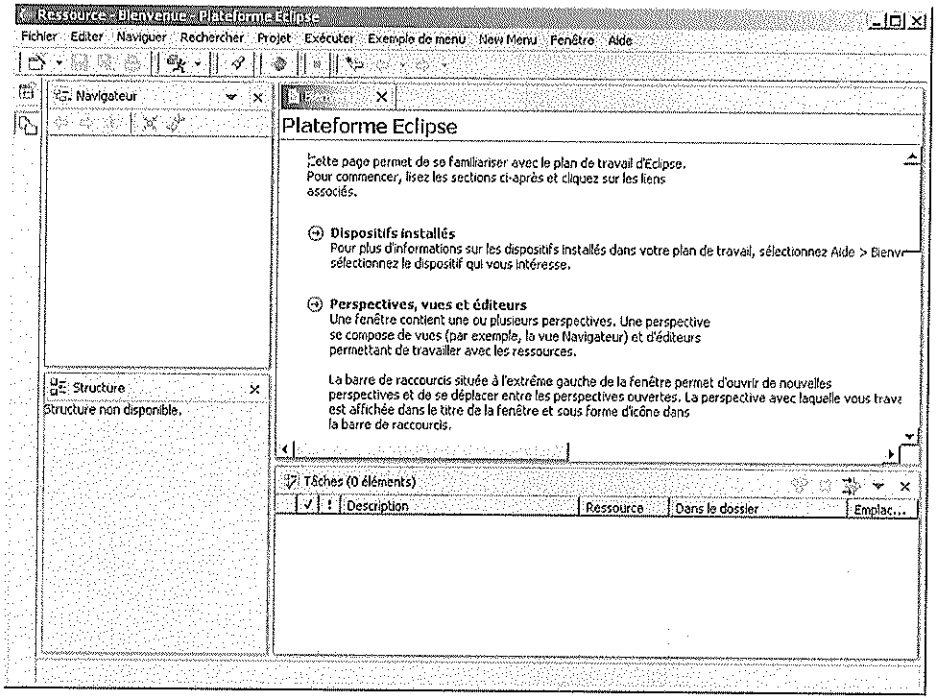


Figure 11-19. Utilisation du nouveau plug-in

Si l'on sélectionne l'option de menu New Menu → New Action ou si l'on clique sur le nouveau bouton, on doit voir la boîte de message New Plug-In (voir figure 11-20), conformément à ce que l'on attendait.

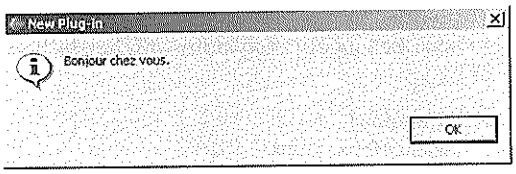


Figure 11-20. Le message affiché grâce au nouveau plug-in

Personnalisation automatique d'une perspective

Le processus visant à personnaliser une perspective, à chaque fois qu'un nouveau plug-in est lancé, au moyen de Fenêtre → Personnaliser la perspective, est quelque peu fastidieux. Celui-ci peut être évité au moyen du point d'extension `org.eclipse.ui.perspectiveExtensions`, dans *plugin.xml*, qui permet d'ajouter un plug-in automatiquement dans une perspective. La modification à effectuer dans *plugin.xml* est si simple qu'elle peut être réalisée directement, en éditant le code XML pour y utiliser le point d'extension `org.eclipse.ui.perspectiveExtensions`. Par exemple, pour ajouter automatiquement notre plug-in dans la perspective Java, on peut utiliser la classe `org.eclipse.ui.javaPerspective` de la façon suivante dans *plugin.xml* :

```
<?xml version="1.0" encoding="UTF-8"?>
<plugin
   id="org.eclipsebook.ch11.Ch1102"
   name="Ch11_02 Plug-in"
   version="1.0.0"
   provider-name="Livre Eclipse"
   class="org.eclipsebook.ch11.Ch11_02.Ch11_02Plugin">

   <runtime>
      <library name="Ch1102.jar"/>
   </runtime>
   <requires>
      <import plugin="org.eclipse.core.resources"/>
      <import plugin="org.eclipse.ui"/>
   </requires>

   <extension
         point="org.eclipse.ui.actionSets">
      <actionSet
            label="New Action Set"
            visible="true"
            id="org.eclipsebook.ch11.Ch11_02.actionSet1">
         <menu
               label="New Menu"
               id="newMenu">
            <separator
                  name="newGroup">
            </separator>
         </menu>
         <action
               label="New Action"
```

```
            tooltip="Bonjour chez vous."
            class="org.eclipsebook.ch11.Ch11_02.NewAction"
            menubarPath="newMenu/newGroup"
            toolbarPath="newGroup"
            id="org.eclipsebook.ch11.Ch11_02.action1">
         </action>
      </actionSet>
   </extension>
   <extension
      point = "org.eclipse.ui.perspectiveExtensions">
      <perspectiveExtension
          targetID="org.eclipse.ui.javaPerspective">
         <actionSet
             id="org.eclipsebook.ch11.Ch11_02.actionSet1">
         </actionSet>
      </perspectiveExtension>
   </extension>
   </plugin>
```

Pour ajouter automatiquement le plug-in à la perspective Ressource, on utilisera plutôt la classe `org.eclipse.ui.resourcePerspective`. Sauvegardons *plugin.xml* après avoir effectué les modifications et relançons le plan de travail d'exécution. Le plug-in devrait être installé au lancement.

CHAPITRE 12

Développement d'un plug-in : création d'éditeurs et de vues

L'objectif de ce chapitre est de créer des éditeurs et des vues personnalisés au moyen de plug-in. Le développement de plug-in est un monde en soi, et la création d'un plug-in commercial peut impliquer le développement de dizaines de fichiers. Fort heureusement, le PDE Eclipse est fourni avec un certain nombre d'assistants qui contribuent à l'écriture de l'architecture du code, économisant au programmeur un temps précieux. On verra dans ce chapitre comment utiliser ces assistants et comment remettre en forme le code généré par eux pour réaliser le plug-in désiré.

Création d'un éditeur multipage

Le premier exemple de ce chapitre va nous amener à créer un éditeur multipage associé à une extension de fichier définie par nous, à savoir *.xyz*. Lorsque l'utilisateur double-cliquera, dans la Vue Packages, sur un fichier possédant cette extension, Eclipse utilisera notre éditeur pour ouvrir le fichier et pour l'éditer. Cet éditeur possédera deux onglets correspondant à ses deux pages d'édition – l'onglet par défaut affichera le contenu du fichier texte, alors que l'onglet Trié permettra d'afficher ce même contenu mais trié dans l'ordre alphabétique. On pourra créer des fichiers d'extension *.xyz* au moyen de l'option de menu Nouveau → Fichier mais le plug-in de notre exemple va aussi posséder un assistant qui créera des fichiers *.xyz* pour l'utilisateur et placera un texte par défaut dans ceux-ci.

Création du code

Pour créer l'architecture générale du code de cet exemple, on sélectionne Nouveau → Projet, puis, dans la boîte de dialogue Nouveau projet, Développement de plug-in (partie gauche de la boîte) et Projet de plug-in (partie droite de la boîte) ; on clique alors Suivant. On nomme le projet `org.eclipsebook.ch12.Ch12_01` dans la page suivante et on clique à nouveau Suivant. On conserve les choix par défaut de la page Structure de projet de plug-in pour faire de ce projet un projet Java. On clique de nouveau Suivant : on obtient alors la page Générateurs de code de plug-in de la figure 12-1.

On sélectionne l'option Plug-in avec éditeur multipage, comme on le voit dans la figure, et on clique Suivant. Dans la page qui suit, on initialise le nom du fournisseur à `Eclipse Book` et on clique Suivant à nouveau. Dans la page qui suit, on initialise

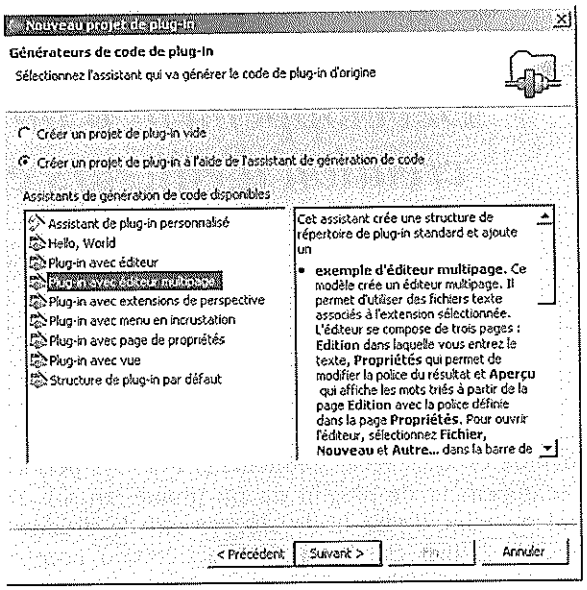


Figure 12-1. Création d'un plug-in comportant un éditeur multipage

l'option Extensions de fichers à xyz pour associer celle-ci au plug-in (voir figure 12-2) ; on clique à nouveau Suivant.

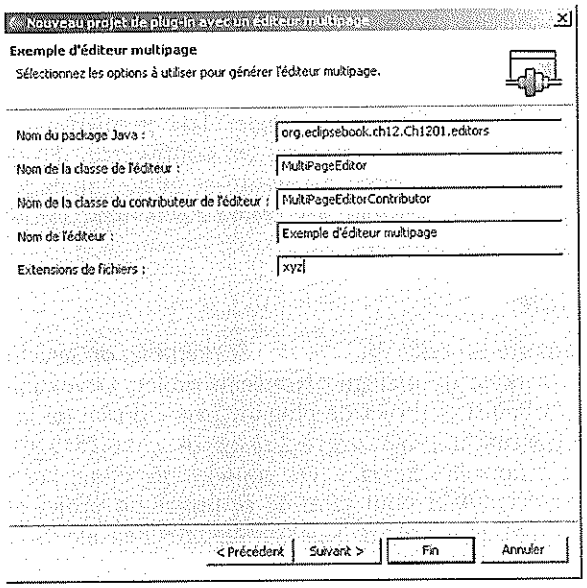


Figure 12-2. Configuration de l'éditeur du plug-in

Dans la page suivante, on initialise l'extension de fichier à xyz, et on donne comme nom de fichier initial `fichier.xyz` (voir figure 12-3). On peut alors créer le squelette du code de ce plug-in en cliquant Fin.

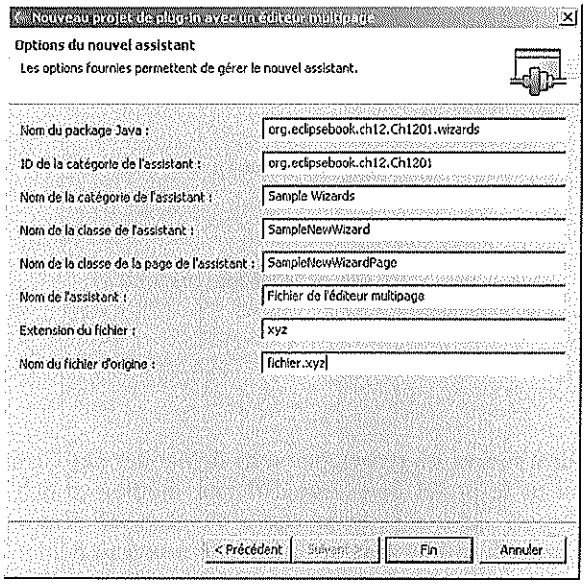


Figure 12-3. Configuration de l'assistant du plug-in

Voici les fichiers créés par l'assistant du PDE et ajoutés par lui dans le répertoire *src*, dans le cadre du projet `org.eclipsebook.ch12.Ch12_01` :

```
src
|_ _org.eclipsebook.ch12.Ch12_01
|    |_ _Ch12_01Plugin.java              Le fichier plug-in standard
|
|_ _org.eclipsebook.ch12.Ch1201.editors
|    |_ _MultiPageEditor.java            Le code de l'éditeur
|    |_ _MultiPageEditorContributor.java Les actions de l'éditeur
|                                        (options de menu/barre d'outils)
|
|_ _org.eclipsebook.ch12.Ch1201.wizards
|    |_ _SampleNewWizard.java            Le code de l'assistant
|    |_ _SampleNewWizardPage.java        Le code de la page de l'assistant
|
|__ _plugin.xml
```

Dissection de l'assistant

Le plug-in supporte un assistant et un éditeur multipage. Nous allons concentrer notre attention sur l'éditeur, mais un examen rapide de l'assistant est tout de même instructif. La mission de cet assistant est de créer un nouveau fichier, *fichier.xyz*, et de l'ajouter dans le projet. Le code Java qui effectue cette tâche est contenu dans la classe `SampleNewWizard`, qui est la fille de la classe `Wizard`. La page affichée par l'assistant est

supportée par la classe SampleNewWizardPage, qui construit un objet SWT composite affichant les contrôles avec lequels l'utilisateur va interagir. Voici comment le code de la classe SampleNewWizard crée la page affichée par l'assistant :

```
public class SampleNewWizard extends Wizard implements INewWizard {
        private SampleNewWizardPage page;
        private ISelection selection;

        public SampleNewWizard() {
                super();
                setNeedsProgressMonitor(true);
        }

        public void addPages() {
                page = new SampleNewWizardPage(selection);
                addPage(page);
        }
        .
        .
        .
```

Lorsque l'utilisateur a fourni les renseignements demandés par l'assistant (i.e. le nom du nouveau fichier et le projet dans lequel il faut ajouter ce fichier) et lorsqu'il a cliqué Fin, la méthode doFinish de la classe SampleNewWizard est appelée. Cette méthode reçoit le nom du projet devant intégrer le nouveau fichier ainsi que le nom de ce fichier. Le code concerné crée le nouveau fichier, l'ajoute dans le projet Eclipse ad hoc et l'ouvre sous Eclipse – ces opérations sont parmi les plus impressionnantes que l'on peut réaliser dans un plug-in :

```
private void doFinish(
        String containerName,
        String fileName,
        IProgressMonitor monitor)
        throws CoreException {
        // create a sample file
        monitor.beginTask("Creating " + fileName, 2);
        IWorkspaceRoot root = ResourcesPlugin.getWorkspace().getRoot();
        IResource resource = root.findMember(new Path(containerName));
        if (!resource.exists() || !(resource instanceof IContainer)) {
                throwCoreException("Container \"" + containerName +
          "\" does not exist.");
        }
        IContainer container = (IContainer) resource;
        final IFile file = container.getFile(new Path(fileName));
        try {
                InputStream stream = openContentStream();
                if (file.exists()) {
                        file.setContents(stream, true, true, monitor);
                } else {
                        file.create(stream, true, monitor);
                }
                stream.close();
        } catch (IOException e) {
        }
```

```
            monitor.worked(1);
            monitor.setTaskName("Opening file for editing...");
            getShell().getDisplay().asyncExec(new Runnable() {
                    public void run() {
                            IWorkbenchPage page =
                                    PlatformUI.getWorkbench()
                        .getActiveWorkbenchWindow().getActivePage();
                            try {
                                    page.openEditor(file);
                            } catch (PartInitException e) {
                            }
                    }
            });
            monitor.worked(1);
    }
```

Le texte par défaut contenu dans le nouveau fichier est initialisé lorsque `doFinish` ouvre un nouveau flux d'entrée pour lire les données texte qui doivent être stockées dans le fichier. Cela est réalisé par la méthode `openContentStream`, que l'on va modifier afin de fournir notre propre texte, « Voici un petit texte. », comme donnée par défaut à insérer dans le nouveau fichier. Voici le nouveau code de cette méthode :

```
    private InputStream openContentStream() {
            String contents = "Voici un petit texte.";
            return new ByteArrayInputStream(contents.getBytes());
    }
```

Ce code modifie l'assistant qui va désormais charger notre texte par défaut dans le nouveau fichier.

Codage de l'éditeur multipage

Le traitement principal de cet exemple est localisé dans *MultiPageEditor.java*, qui fournit l'implémentation de l'éditeur de fichiers *.xyz* (*MultiPageEditorContributor.java* contribue aussi à l'implémentation de l'éditeur mais de façon périphérique pour notre propos, car il fournit le support pour les actions sur le menu et la barre d'outils, ce qui a été discuté dans le chapitre 11). L'objectif présent est d'ajouter deux pages dans cet éditeur : la première permettant simplement d'éditer le texte contenu dans un fichier *.xyz*, la seconde affichant le texte trié dans l'ordre alphabétique. Pour mettre en place ces deux pages, nous allons adapter le code écrit pour nous par l'assistant du PDE.

Dans *MultiPageEditor.java*, la classe `MultiPageEditor` implémente l'éditeur créé par nous (celle-ci est la classe fille de `MultiPageEditorPart`, qui est une classe Eclipse utilisant un système d'onglets fourni par SWT pour afficher les pages de l'éditeur). Pour afficher le texte dans la première page de l'éditeur, le code de cet exemple utilise une instance de la classe `org.eclipse.editors.ui.text.TextEditor`, nommée `editor` ; pour afficher le texte trié dans la seconde page, le code utilise une instance de la classe `org.eclipse.editors.custom.StyledText` nommée `text`. Une fois ces objets créés, on peut utiliser la méthode `addPage` de la classe `MultiPageEditorPart` pour ajouter les deux nouvelles pages à l'éditeur. Pour ce faire, il faut surcharger la méthode `createPages` de la classe `MultiPageEditor`, qui est en charge de la création des pages dans l'éditeur et qui est appelée automatiquement au démarrage de l'éditeur. Voici

notre code modifié et adapté :

```
public class MultiPageEditor extends MultiPageEditorPart {

        private TextEditor editor;

        private StyledText text;

        public MultiPageEditor() {
                super();
        }

        void createPage0() {
                try {
                        editor = new TextEditor();
                        int index = addPage(editor, getEditorInput());
                        setPageText(index, editor.getTitle());
                } catch (PartInitException e) {
                        ErrorDialog.openError(
                                getSite().getShell(),
                                "Error creating nested text editor",
                                null, e.getStatus());
                }
        }

        void createPage1() {
                Composite composite = new Composite(getContainer(), SWT.NONE);
                FillLayout layout = new FillLayout();
                composite.setLayout(layout);
                text = new StyledText(composite, SWT.H_SCROLL | SWT.V_SCROLL);
                text.setEditable(false);

                int index = addPage(composite);
                setPageText(index, "Trié");
        }

        protected void createPages() {
                createPage0();
                createPage1();
        }
```

Cela effectue la création et l'insertion des deux pages que nous allons utiliser dans l'éditeur. L'étape suivante consiste à activer des options telles que Fichier → Sauvegarder ou Fichier → Sauvegarder sous lorsque l'utilisateur les sélectionne dans le système de menu standard d'Eclipse ou dans la barre d'outils. Pour ce faire, il faut surcharger les méthodes `doSave` et `doSaveAs`, en utilisant la méthode `getEditor` pour obtenir l'objet de type `IEditorPart` qui effectue réellement cette tâche, et en appelant les méthodes homonymes de cet objet. Le code concerné est construit automatiquement par l'assistant de PDE :

```
public void doSave(IProgressMonitor monitor) {
        getEditor(0).doSave(monitor);
}
```

```
public void doSaveAs() {
        IEditorPart editor = getEditor(0);
        editor.doSaveAs();
        setPageText(0, editor.getTitle());
        setInput(editor.getEditorInput());
}

public boolean isSaveAsAllowed() {
        return true;
}
```

On doit encore trier les mots du texte en vue de leur affichage dans la seconde page. Nous allons gérer cela en surchargeant la méthode pageChange, et si la nouvelle page a pour mission d'afficher le texte trié (l'éditeur standard est la page 0, l'éditeur de texte trié étant la page 1), on triera les mots du texte avant leur affichage au moyen d'une méthode spécifique, sort :

```
protected void pageChange(int newPageIndex) {
        super.pageChange(newPageIndex);
        if (newPageIndex == 1) {
                sort();
        }
}
```

Dans la méthode sort, on accède au texte de l'éditeur de base (objet editor) de la façon suivante : String unsortedText = editor.getDocumentProvider( ).getDocument (editor.getEditorInput()).get().

L'étape suivante consiste à trier ce texte en le stockant tout d'abord sous la forme d'un ArrayList de mots au moyen de la classe StringTokenizer, les séparateurs étant les espaces, les tabulations, les retours chariots et les passages à la ligne. Voici le code concerné :

```
void sort() {
        String unsortedText = editor.getDocumentProvider()
         .getDocument(editor.getEditorInput()).get();

        StringTokenizer stringTokenizer =
                new StringTokenizer(unsortedText, " \t\n\r");
        ArrayList wordArrayList = new ArrayList();
        while (stringTokenizer.hasMoreTokens()) {
                wordArrayList.add(stringTokenizer.nextToken());
        }
    .
    .
    .
```

Après avoir décomposé le texte en mots et stocké ceux-ci dans un ArrayList, il faut les trier puis les réassembler afin d'obtenir un texte contenant tous ces mots triés. L'étape finale a pour but de stocker le résultat de ce traitement dans un contrôle de type StyleText, nommé text, qui affiche le texte trié dans la seconde page de l'éditeur. Voici le code :

```
void sort() {
        String unsortedText = editor.getDocumentProvider()
         .getDocument(editor.getEditorInput()).get();
```

```
        StringTokenizer stringTokenizer =
                new StringTokenizer(unsortedText, " \t\n\r");
        ArrayList wordArrayList = new ArrayList();
        while (stringTokenizer.hasMoreTokens()) {
                wordArrayList.add(stringTokenizer.nextToken());
        }

        Collections.sort(wordArrayList, Collator.getInstance());
        StringWriter sortedText = new StringWriter();
        for (int loopIndex = 0; loopIndex < wordArrayList.size(); loopIndex++) {
                sortedText.write(((String) wordArrayList.get(loopIndex)));
                sortedText.write(System.getProperty("line.separator"));
        }

        text.setText(sortedText.toString());
}
```

Ce code permet de trier et d'afficher le texte de *fichier.xyz* dans l'éditeur de l'onglet Trié, ce qui achève le code personnalisé de *MultiPageEditor.java*. Pour faire fonctionner cet éditeur personnalisé, on démarre le plan de travail d'exécution et l'on crée un nouveau projet Java, `testProjet`. On clique le bouton droit de la souris sur testProjet et l'on sélectionne Nouveau → Autre pour ouvrir la boîte de dialogue Nouveau de la figure 12-4. On sélectionne ensuite l'option Sample Wizards dans la zone de gauche et l'option Fichier de l'éditeur multipage dans la zone de droite, puis l'on clique Suivant pour ouvrir le nouvel assistant.

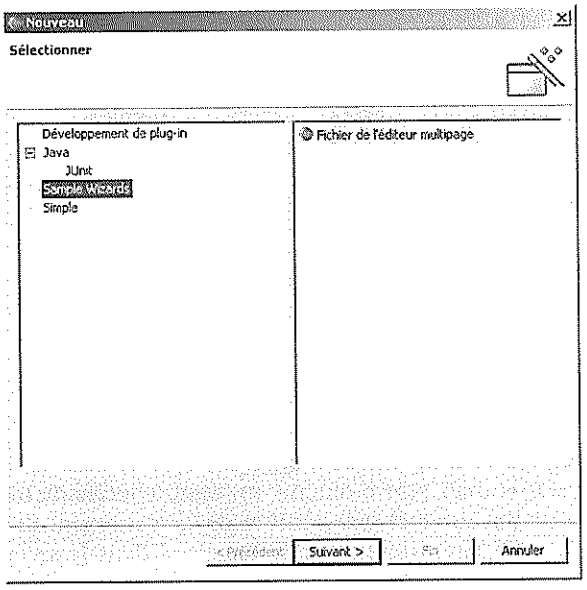


Figure 12-4. Invocation du nouvel assistant

Notre nouvel assistant indique le nom du nouveau fichier, *fichier.xyz* par défaut, et demande que l'on spécifie un conteneur pour le fichier (voir figure 12-5). On accède à testProjet au moyen du bouton Browse et l'on clique Fin pour créer le fichier.

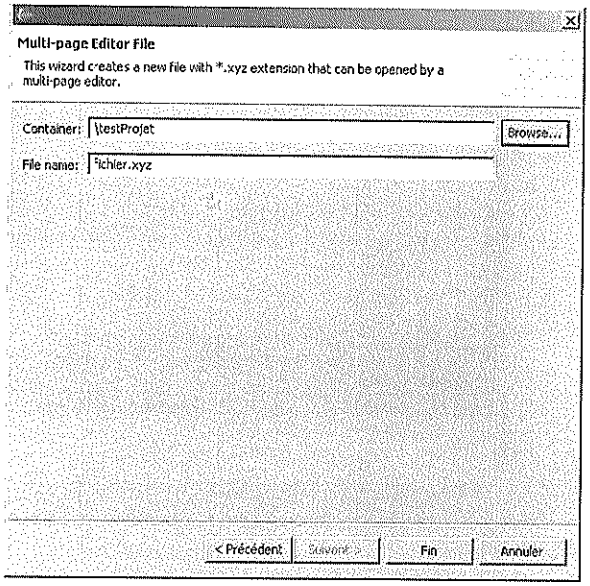


Figure 12-5. Utilisation du nouvel assistant

Cela crée le fichier *fichier.xyz* et procède à son ouverture pour l'édition du texte spécifié plus haut (voir figure 12-6). On notera que *fichier.xyz* est aussi visible dans la Vue Packages.

Notre éditeur possède deux onglets : fichier.xyz et Trié. Cliquer sur le second onglet permet d'accéder au second éditeur, qui visualise le texte trié (voir figure 12-7).

Notre éditeur personnalisé fonctionne bien comme attendu – il ouvre tout fichier *.xyz* double-cliqué dans la Vue Packages. L'utilisateur peut effectuer des modifications dans tout fichier ainsi ouvert grâce à l'éditeur de la première fenêtre et sauvegarder celui-ci au moyen des options Fichier → Sauvergarder et Fichier → Sauvegarder sous, sous Eclipse, ou des boutons correspondants de la barre d'outils. Notre propre éditeur est donc bien complètement intégré dans Eclipse.

Création d'une vue

Dans l'exemple qui suit, on va utiliser un plug-in pour créer une nouvelle vue. On commence par créer un nouveau projet de plug-in nommé `org.eclipsebook.ch12.Ch12_02`. Dans la page Générateurs de code de plug-in de la figure 12-8, on sélectionne l'assistant Plug-in avec vue et on clique Suivant.

Dans la page suivante, on tape le nom du fournisseur (Eclipse Book) et l'on clique sur Suivant pour accéder à la boîte de dialogue Principaux paramètres de la vue, reproduite dans la figure 12-9. Dans celle-ci, on donne le nom de la vue et sa catégorie – on

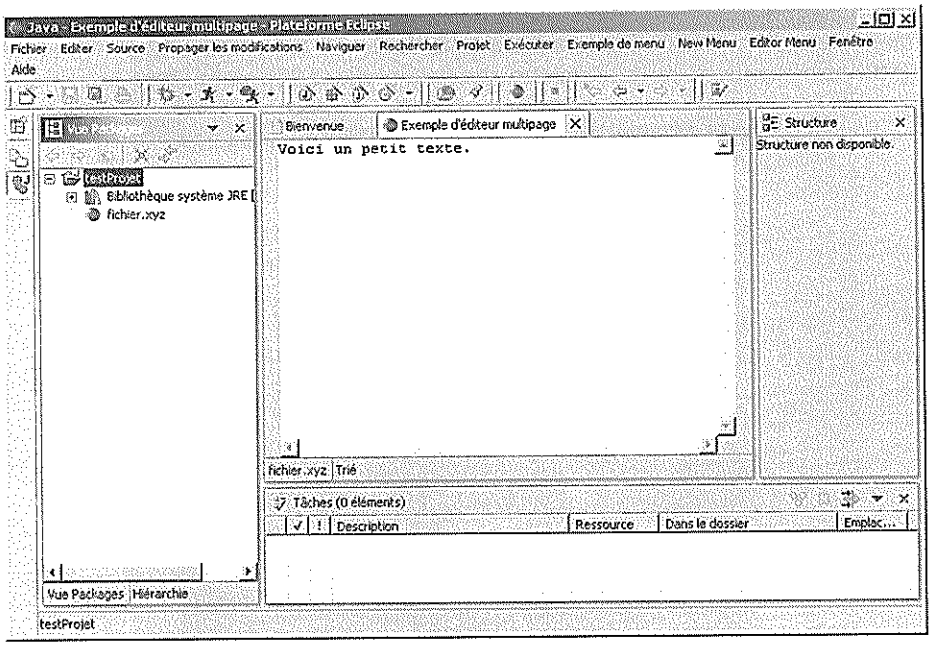


Figure 12-6. Edition de fichier.xyz

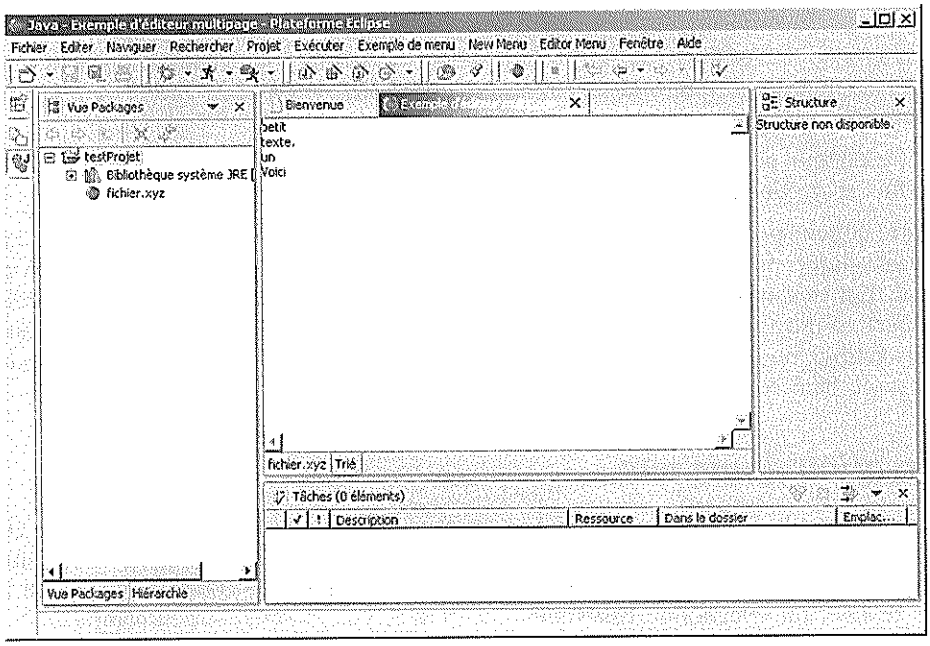


Figure 12-7. Visualisation du texte trié

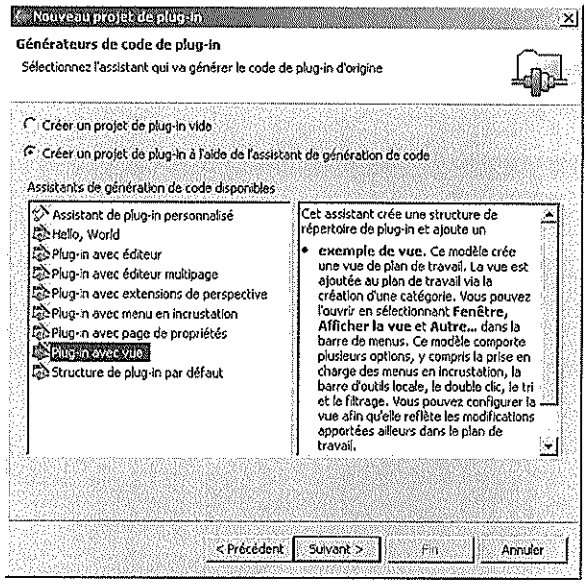


Figure 12-8. Création d'un plug-in comportant une vue

conserve ici les propositions par défaut, qui permettront de construire une vue de type afficheur de table (i.e. les éléments de la vue seront affichés dans une table), nommée Exemple de vue et appartenant à la catégorie Exemple de catégorie. On clique à nouveau Suivant pour accéder à la dernière page de l'assistant.

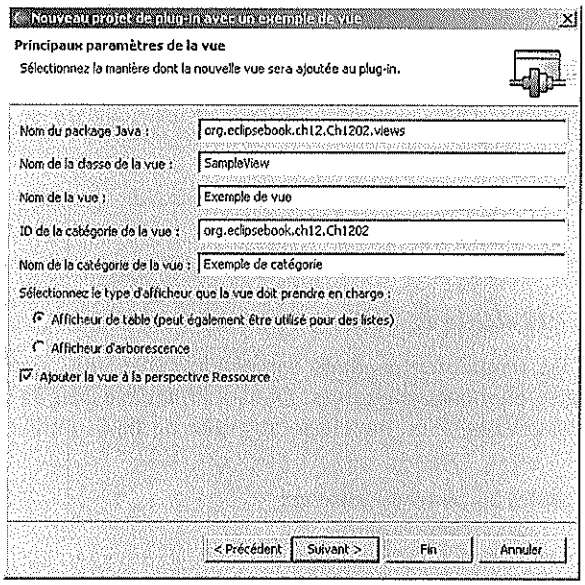


Figure 12-9. Configuration de la vue

La dernière page, présentée dans la figure 12-10, permet de configurer les actions de la vue, telles que l'action à effectuer lorsque l'utilisateur double-clique sur un élément de la vue. On conserve les choix par défaut et l'on clique sur Fin pour créer la structure de ce plug-in.

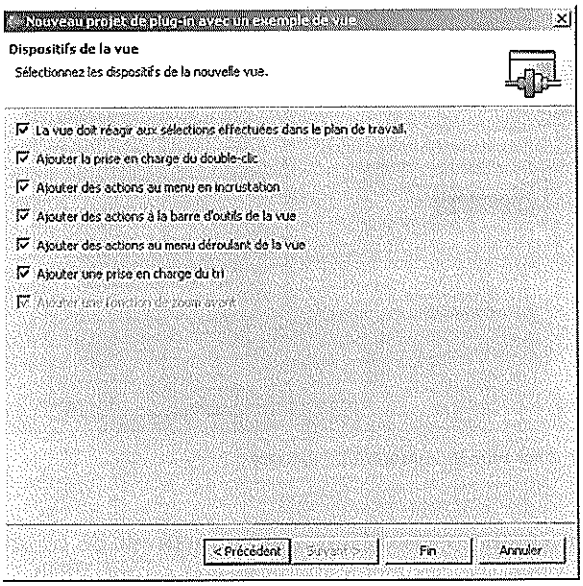


Figure 12-10. Configuration des actions de la vue

Voici les fichiers créés et ajoutés dans le répertoire *src* du projet :

```
src
|_ _org.eclipsebook.ch12.Ch12_02
|   |_ _Ch12_02Plugin.java              Le fichier standard de plug-in
|
|_ _org.eclipsebook.ch12.Ch1202.views
|   |_ _SampleView.java                 Le fichier de la vue
|
|__ _plugin.xml
```

Ajout d'éléments dans le vue

La nouvelle vue est implémentée dans *SampleView.java*, que l'on va adapter afin qu'elle affiche quatre éléments cliquables. Dans *SampleView.java*, la classe `SampleView` est la fille de la classe `ViewPart`, une classe abstraite de base pour les vues gérées par plug-in :

```
public class SampleView extends ViewPart {
        private TableViewer viewer;
        private Action action1;
        private Action action2;
        private Action doubleClickAction;
        .
```

```
.
.
```

Eclipse obtient le contenu d'une vue au moyen de méthodes définies dans l'interface IStructuredContentProvider. On va adapter le code de la classe, nommée ViewContentProvider et implémenter celle-ci, pour que la vue affiche quatre éléments (« Elément 1 », « Elément 2 », etc). Voici le code :

```
class ViewContentProvider implements IStructuredContentProvider {
        public void inputChanged(Viewer v, Object oldInput, Object newInput) {
        }
        public void dispose() {
        }
        public Object[] getElements(Object parent) {
                return new String[] { "Elément 1", "Elément 2", "Elément 3",
                        "Elément 4" };
        }
}
```

La vue est créée lorsqu'Eclipse appelle la méthode createPartControl. Voici comment le code utilise notre classe ViewContentProvider pour obtenir le contenu de la vue :

```
public void createPartControl(Composite parent) {
        viewer = new TableViewer(parent, SWT.MULTI | SWT.H_SCROLL |
                        SWT.V_SCROLL);
        viewer.setContentProvider(new ViewContentProvider());
        viewer.setLabelProvider(new ViewLabelProvider());
        viewer.setSorter(new NameSorter());
        viewer.setInput(ResourcesPlugin.getWorkspace());
        makeActions();
        hookContextMenu();
        hookDoubleClickAction();
        contributeToActionBars();
}
```

Cela permet ainsi de créer les quatre éléments (« Elément 1 », « Elément 2 », etc.) dans la vue. Ceux-ci apparaîtront dans une table, l'un au-dessous de l'autre. L'étape suivante consiste à programmer une action à déclencher lorsque l'utilisateur interagira avec l'un des éléments.

Configuration des actions pour la vue

Le reste du code de *SampleView.java* permet d'implémenter les actions de la vue à déclencher lorsque l'utilisateur clique, double-clique ou clique le bouton droit de la souris sur un élement de la vue. Nous allons modifier le code de la méthode makeActions pour personnaliser les actions de la vue, afin d'afficher les messages appropriés lorsqu'un élément est sélectionné ou double-cliqué ; on accède à l'objet correspondant à l'élément ainsi sollicité de la façon suivante : `Object obj = ((IStructuredSelection)selection).getFirstElement( )`. Voici le code modifié, qui va déterminer l'élément sur lequel l'utilisateur a agi et afficher un message précisant le nom de l'élément sollicité :

```
private void makeActions() {
        action1 = new Action() {
```

```
                public void run() {
                        ISelection selection = viewer.getSelection();
                        Object obj =
                   ((IStructuredSelection)selection).getFirstElement();
                        showMessage("Vous avez sélectionné " + obj.toString());
                }
        };
        action1.setText("Action 1");
        action1.setToolTipText("Action 1 tooltip");
        action1.setImageDescriptor(PlatformUI.getWorkbench().getSharedImages().
                getImageDescriptor(ISharedImages.IMG_OBJS_INFO_TSK));

        action2 = new Action() {
                public void run() {
                        showMessage("Action 2 executed");
                }
        };
        action2.setText("Action 2");
        action2.setToolTipText("Action 2 tooltip");
        action2.setImageDescriptor(PlatformUI.getWorkbench().getSharedImages().
                getImageDescriptor(ISharedImages.IMG_OBJS_TASK_TSK));
        doubleClickAction = new Action() {
                public void run() {
                        ISelection selection = viewer.getSelection();
                        Object obj =
                   ((IStructuredSelection)selection).getFirstElement();
                        showMessage("Vous avez double-cliqué sur " +
                          obj.toString());
                }
        };
}
```

Comme on peut le voir, le code généré par l'assistant du PDE est conçu de telle façon qu'il est aisément modifiable. Pour voir la nouvelle vue à l'œuvre, on démarre le plan de travail d'exécution, on sélectionne Fenêtre → Afficher la vue → Autre, puis l'option Exemple de vue dans le dossier Exemple de catégorie (voir figure 12-11) ; on clique enfin OK pour afficher la nouvelle vue.

La nouvelle vue apparaît dans la figure 12-12, dans laquelle on peut voir les quatre éléments personnels en bas de la fenêtre Eclipse.

Si l'on clique le bouton droit de la souris sur un élément et si l'on sélectionne l'action 1, le code associé est exécuté et la boîte de dialogue de la figure figure 12-13, qui indique l'identité de l'élément sélectionné, est affichée.

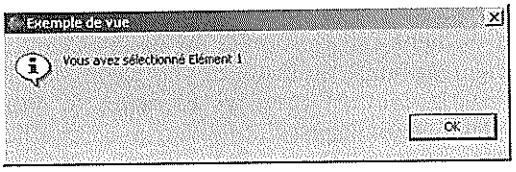


Figure 12-13. Sélection d'un élément

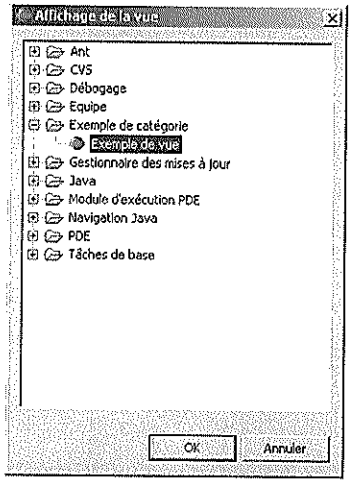


Figure 12-11. Affichage de la nouvelle vue

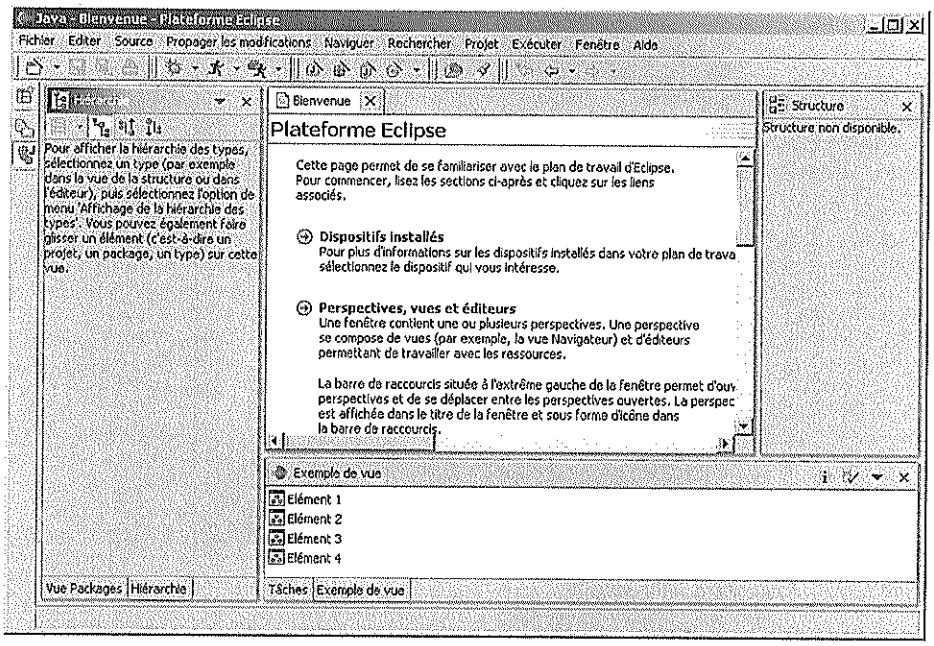


Figure 12-12. La nouvelle vue

De même, si l'on double-clique sur un élément, la boîte de dialogue de la figure 12-14 s'affiche aussi en indiquant l'identité de l'élément double-cliqué.

Le résultat est bien conforme aux attentes – une nouvelle vue a été créée, qui réagit aux actions de l'utilisateur sur les éléments de celle-ci. On a ainsi créé un plug-in comportant des vues, des éditeurs et même des assistants. Les bases de ce type de développement ayant été présentées, il nous faut maintenant décrire les méthodes de déploiement de plug-in.

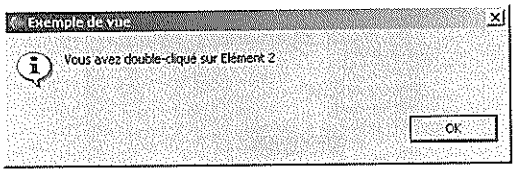


Figure 12-14. Double-clic sur un élément

Déploiement d'un plug-in

La réalisation de paquetages de déploiement de plug-in est relativement simple sous Eclipse. Pour ce faire, il suffit de sélectionner Fichier → Exporter, puis l'option Plug-in et fragments déployables dans la boîte de dialogue Exporter de la figure 12-15.

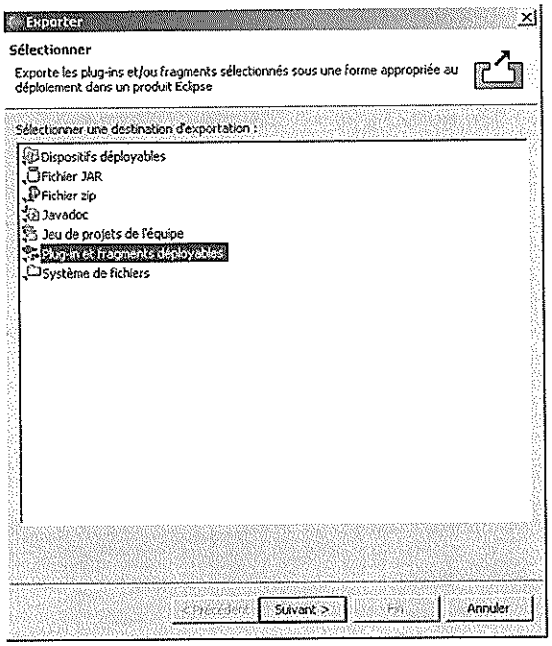


Figure 12-15. Deploiement d'un plug-in

Cliquons Suivant pour ouvrir la page Plug-in et fragments déployables de la figure 12-16. Un plug-in peut être déployé sous la forme de fichiers *.jar* ou *.zip* ; dans le cas présent, on va créer un fichier *.zip*. Pour ce faire, on tape le nom complet du fichier *.zip* à créer – *plugins.zip* – dans la zone de texte éditable Nom du fichier et l'on sélectionne tous les projets de plug-in de ce chapitre.

Cela permet d'inclure les deux plug-in de ce chapitre (y compris les fichiers *plugin.xml* et *.jar*, pour chacun d'entre eux) dans le fichier *plugins.zip*. La décompression de ce ficher dans le répertoire de distribution d'Eclipse provoque l'installation des plug-in – les deux fichiers de ce premier plug-in seront stockés dans *eclipse\plugins*

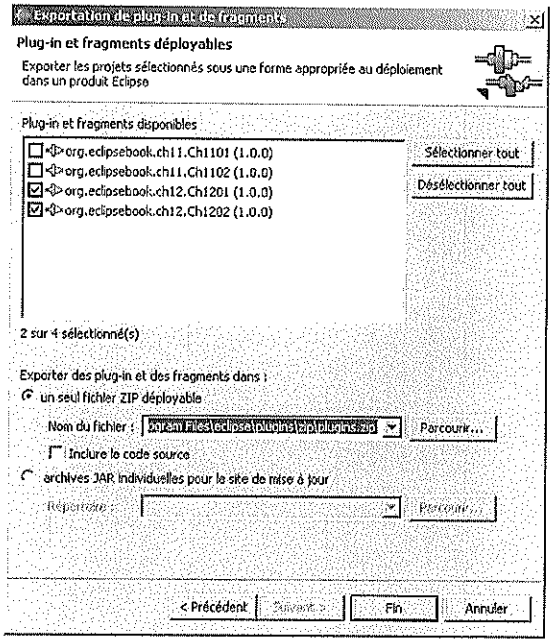


Figure 12-16. Création d'un fichier .zip de plug-in

\org.eclipsebook.ch12.Ch1201_1.0.0, alors que ceux du second le seront dans *eclipse\plugins\org.eclipsebook.ch12.Ch1202_1.0.0*.

CHAPITRE 13

Eclipse 3.0

Eclipse 3.0 est en cours de développement : ce chapitre a pour but de présenter les grandes innovations de cette nouvelle version, considérée à ce jour comme une version bêta.

L'exposé s'appuie sur la version bêta la plus récente d'Eclipse 3.0. Eclipse 3.0 est conçue de façon compatible avec les versions 2.0 et 2.1, dans la mesure du possible. D'après l'équipe en charge de cette mise à jour, « un guide exhaustif de portage vers Eclipse 3.0 couvrira toutes les modifications sensibles de l'API et décrira les méthodes de portage des plug-in de la version 2.1 vers la version 3.0 ». Les projets des versions 2.0 et 2.1 seront importables dans la version 3.0, mais une boîte de dialogue précisera que ces projets devront être modifiés et qu'il n'y aura pas de compatibilité du résultat avec les versions antérieures. Il conviendra donc de sauvegarder ses projets avant toute modification sous Eclipse 3.0.

Dans l'état actuel des choses, Eclipse n'a pas fait l'objet d'un portage en langue française. Dans ce chapitre, toutes les commandes et tous les messages figurent donc en anglais dans le texte, leur traduction étant précisée entre parenthèses.

Un survol d'Eclipse 3.0

Dans l'état actuel des choses, Eclipse 3.0 ressemble beaucoup à la version 2.1, comme on peut le voir dans la figure 13-1.

Fondamentalement, les utilisateurs d'Eclipse 2.0 n'auront aucune difficulté à utiliser la version 3.0, même si quelques vues nouvelles et quelques boutons ou choix de menu font leur apparition dans cette nouvelle version.

Cela étant, l'équipe de développement d'Eclipse expérimente de nouveaux *looks* concernant les vues, les éditeurs et les perspectives d'Eclipse 3.0. L'interface de la figure 13-1 pourrait donc évoluer de façon notable.

Les modifications envisagées par l'équipe Eclipse incluent l'utilisation d'onglets aux bords arrondis à la place de ceux (à bords

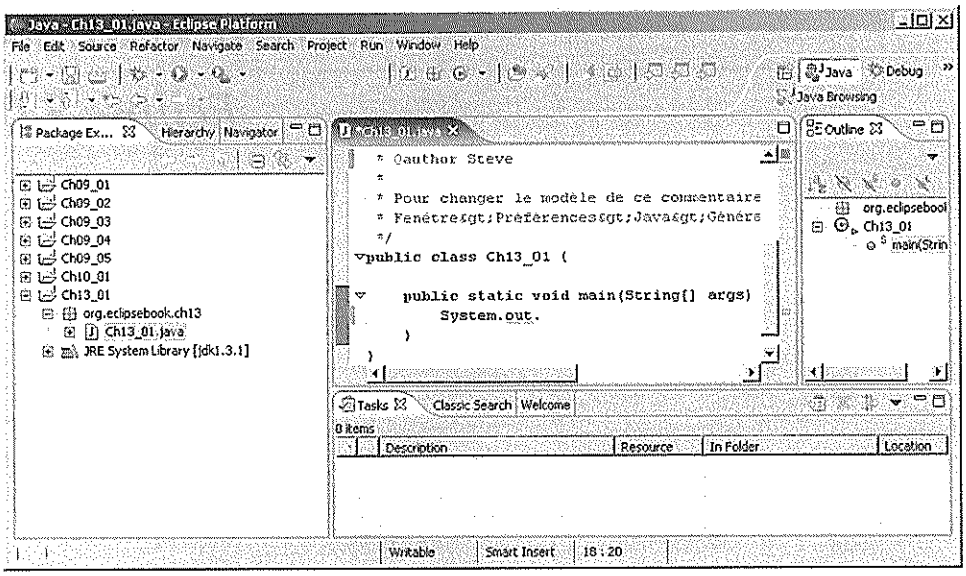


Figure 13-1. La version actuelle d'Eclipse 3.0

carrés) actuellement utilisés, et aussi la présentation d'un seul onglet d'éditeur à la fois, sans empilement (on pourrait accéder aux autres éditeurs au moyen de boutons décorés de flèches), ainsi que l'affichage d'icônes dans la partie supérieure droite des perspectives (et non plus la partie gauche).

Création d'un projet Java

Pour les programmeurs Java, la tâche fondamentale est la création d'un projet Java. Comme on va le voir, le processus de création est (en l'état actuel des choses) virtuellement identique à celui de la version 2.1.

Pour créer un nouveau projet, il suffit de cliquer le bouton droit de la souris sur la vue Navigator (Navigateur) et de sélectionner New (Nouveau) → Project (Projet), ce qui provoque l'ouverture de la boîte de dialogue New project (Nouveau projet). On sélectionne ensuite l'option Java dans la zone de gauche ainsi que Java Project dans la zone de droite, et l'on clique Next. La boîte de dialogue de la figure 13-2 ressemble en tout point à ce que l'on a vu précédemment. On nomme ce projet `Ch13_01` et on clique Next (Suivant).

Cela nous amène à la page familière de la figure 13-3, dans laquelle on définit les caractéristiques du projet, telles que le répertoire contenant le code source, ainsi que les ordres d'importation et d'exportation – toutes ces options sont déjà familières. On clique Finish (Fin) pour créer le projet.

Eclipse 3.0 demande alors si l'on veut basculer vers la perspective Java ; cliquons OK. La nouvelle perspective Java ressemble beaucoup à ce que l'on a vu précédemment (voir figure 13-4).

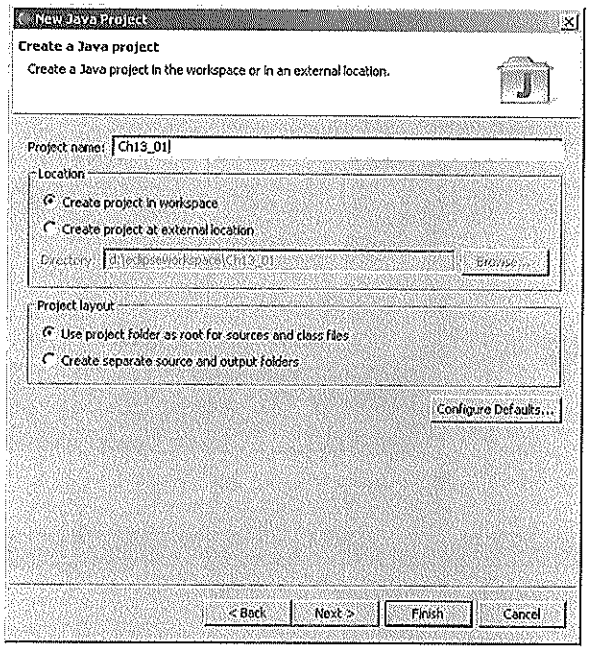


Figure 13-2. Initialisation du nom du nouveau projet

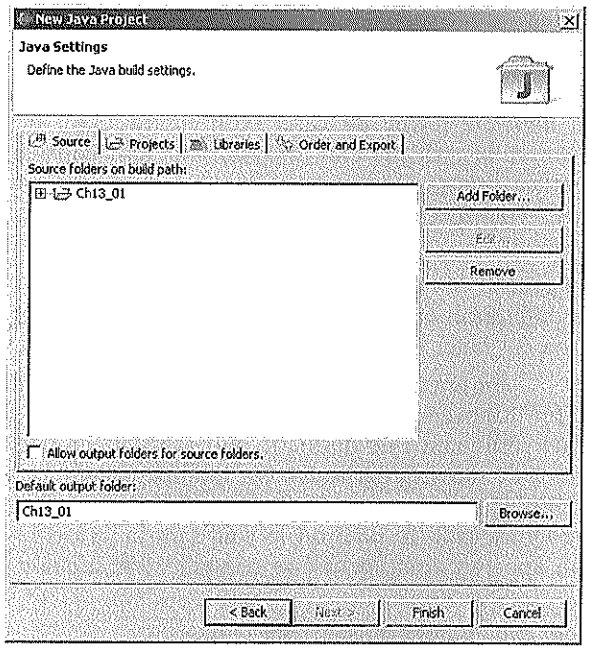


Figure 13-3. Spécification des options du projet

Pour créer une nouvelle classe, il faut cliquer le bouton droit de la souris sur le projet et sélectionner New (Nouveau) → Class (Classe), ce qui ouvre la boîte de dialogue

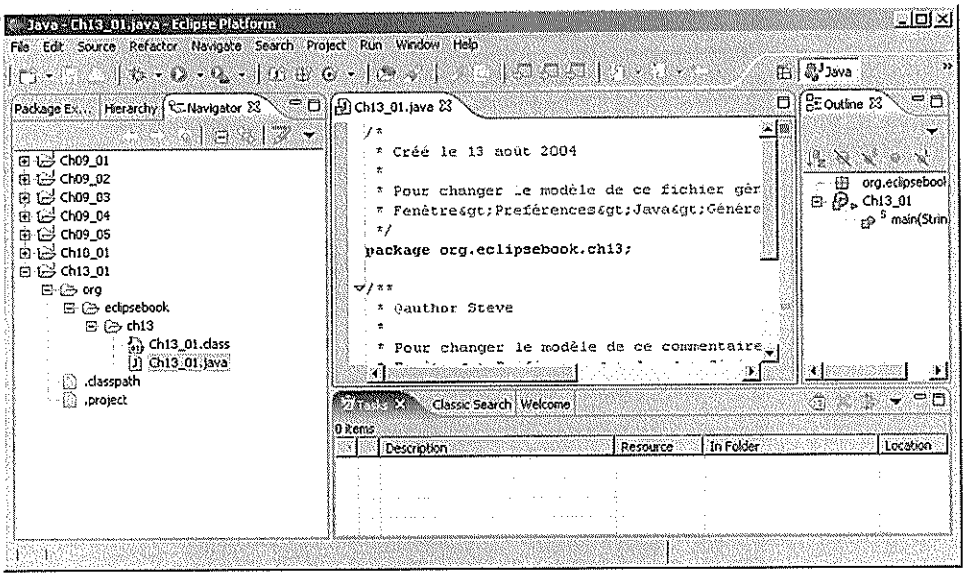


Figure 13-4. La perspective Java de la version 3.0 d'Eclipse

New Java Class (Nouvelle classe Java) de la figure 13-5. C'est bien la même que celle utilisée depuis le début de cet ouvrage ; on donne le nom de cette nouvelle classe, `Ch13_01`, ainsi que son paquetage d'appartenance, `org.eclipsebook.ch13`, et on clique Finish (Fin).

L'ouverture du nouveau fichier Java dans l'éditeur Java révèle quelques changements – par exemple, la nouvelle forme du caret (voir figure 13-6). Il s'agit là du caret *d'insertion habile* (smart insert), présenté un peu plus loin dans ce chapitre (voir la section *Mode d'insertion habile*). On notera aussi deux nouveaux boutons dans la vue Outline (Structure) – Hide Local Types (Cacher les types locaux) et Link With Editor (Lier à l'éditeur).

On trouvera sur la gauche de l'éditeur JDT la barre Quick Diff, qui permet de traquer ligne par ligne les différences entre la version courante et la dernière sauvegarde (ou, si la configuration a été faite, entre la version courante et celle du référentiel CVS).

Pour observer comment fonctionne Quick Diff, on tape `System.out.println("Bienvenue");` dans la méthode `main`, et on sauvegarde le fichier (seule l'icône contenant une disquette apparaît dans la barre d'outils, et non plus celle permettant de sauvegarder tout). On modifie ensuite cette ligne en tapant `System.out.println("Bienvenue dans Eclipse 3.0");` et on place le curseur de la souris sur la barre Quick Diff, comme on le voit dans la figure 13-7. La version originale du code qui apparaît alors, peut être récupérée en cliquant dessus.

Quick Diff est aussi utilisable par les plug-in.

Pour exécuter ce code, on sélectionne Run (Exécuter) → Run As (Exécuter en tant que) → Java Application (Application Java) – noter la nouvelle option de menu Run

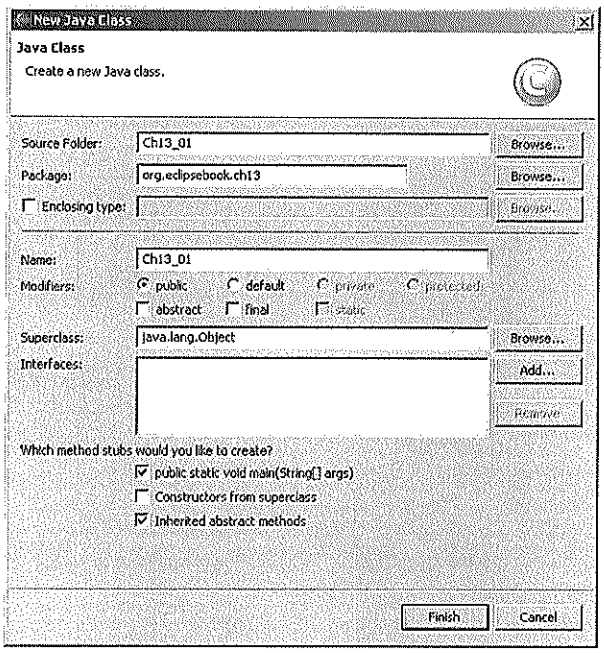


Figure 13-5. Création d'une nouvelle classe

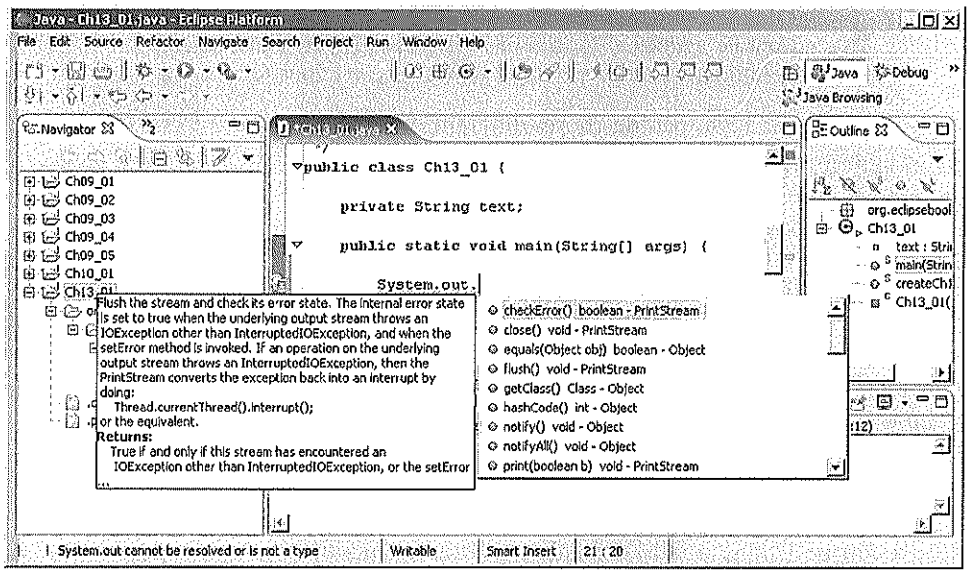


Figure 13-6. Utilisation de l'éditeur Java

(Exécuter) → Run As (Exécuter en tant que) → JUnit Plug-in Test (unité de test de plug-in JUnit) –, ce qui donne le résultat de la figure 13-8, dans la vue Console. On peut constater que le processus de création et d'exécution d'une application Java sous Eclipse 3.0 est très proche de celui de la version 2.1.

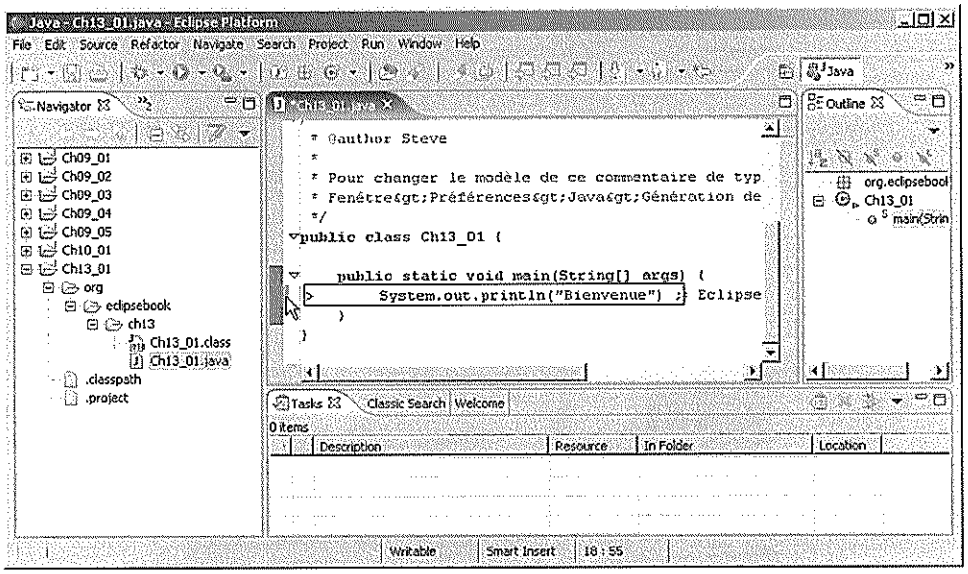


Figure 13-7. Utilisation de Quick Diff

On notera la présence de nouveaux boutons dans la vue Console : Pin Console (Bloquer la console), Display Selected Console (Afficher la console sélectionnée), et Remove All Terminated Launches (Effacer toutes les exécutions terminées).

On va voir que là aussi, peu de choses diffèrent entre les versions 2.1 et 3.0 d'Eclipse.

La modification la plus importante de la version 3.0 est que la plateforme Eclipse a été élargie pour constituer une plateforme d'accueil générale, conçue pour fonctionner avec n'importe quel type d'application, et non plus seulement avec des EDI. À partir de la version 3.0, toute application pourra intégrer des éléments nouveaux dans la barre de menu et dans la barre d'outils, le modèle de données supposé n'étant plus nécessairement centré sur l'espace de travail. On en saura un peu plus à l'approche du lancement officiel de la version 3.0.

On notera des modifications spécifiques de diverses parties d'Eclipse – la plateforme Eclipse elle-même, le JDT, le SWT, etc. – que l'on examinera au cas par cas.

Modifications dans la plateforme Eclipse

La plateforme Eclipse a fait l'objet de nombreuses modifications dans sa version 3.0. La plupart d'entre elles ne sont pas visibles – par exemple, lorsque des ressources sont modifiées, une recompilation automatique est déclenchée en tâche de fond, et il existe maintenant une vue Progress (Progression) permettant de tracer ces modifications. Des fonctionnalités désactivées peuvent maintenant être désinstallées, afin d'économiser de l'espace disque – il suffit de sélectionner Help (Aide) → Software Updates (Mises à jour de logiciel) → Manage Configuration (Gestion de la configuration), d'enfoncer le filtre Show disabled feature (Afficher les éléments désactivés) et de

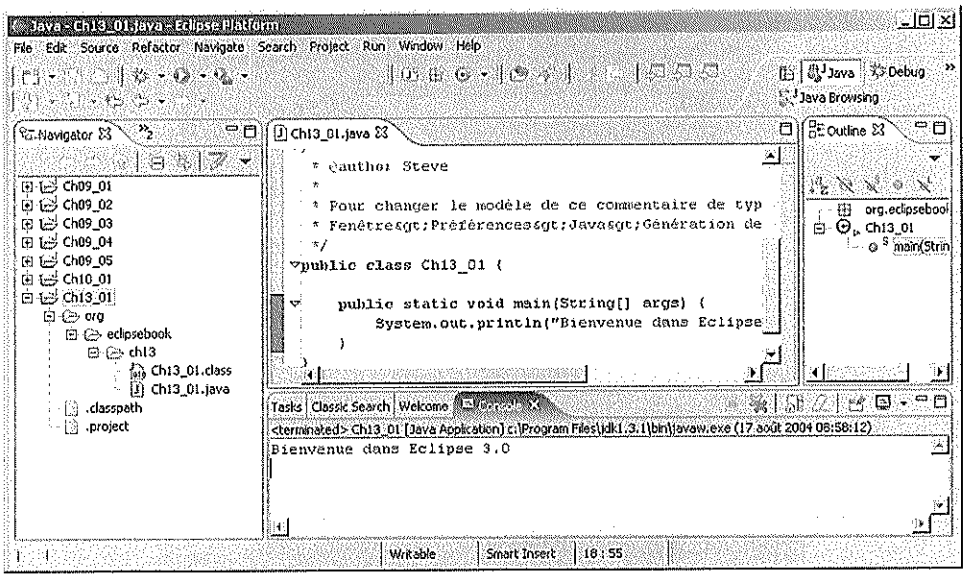


Figure 13-8. Exécution de code

sélectionner Uninstall (Désinstaller) dans le menu contextuel de la fonctionnalité désactivée pour la désinstaller.

Eclipse peut aussi vérifier automatiquement l'existence de mise à jour – pour cela, il suffit d'accéder à la page Window (Fenêtre) → Preferences (Préférences) → Install/Update (Installation/Mise à jour) → Automatic Updates (Mises à jour automatiques).

Certains changements sont cependant plus évidents. L'un des plus utiles est la possibilité d'utiliser des expressions régulières dans la page File search (Recherche de fichier). Il suffit pour cela de sélectionner Search (Rechercher) → File (Fichier) pour ouvrir la boîte de dialogue de la figure 13-9 – noter la nouvelle case à cocher « Regular expression » (Expression régulière) ainsi que la présence d'un assistant.

Pour se familiariser avec les expressions régulières, il suffit de cliquer la case à cocher « Regular expression » (Expression régulière) et de presser la touche F1 pour obtenir une description générale sur l'utilisation de celles-ci pour la recherche de texte.

On peut aussi éditer des fichiers hors de l'espace de travail, au moyen de l'option de menu File (Fichier) → Open External File (Ouvrir un fichier externe), qui ouvre la boîte de dialogue de la figure 13-10. On sélectionne alors un fichier et l'on clique Open (Ouvrir) pour l'ouvrir sous un éditeur externe.

Les annotations qui apparaissent dans divers éditeurs peuvent aussi être élaborées sous Eclipse 3.0. Sous la version 2.1, il existe six types d'annotation qui peuvent être personnalisées (par exemple, les problèmes et les warnings), mais dans la page

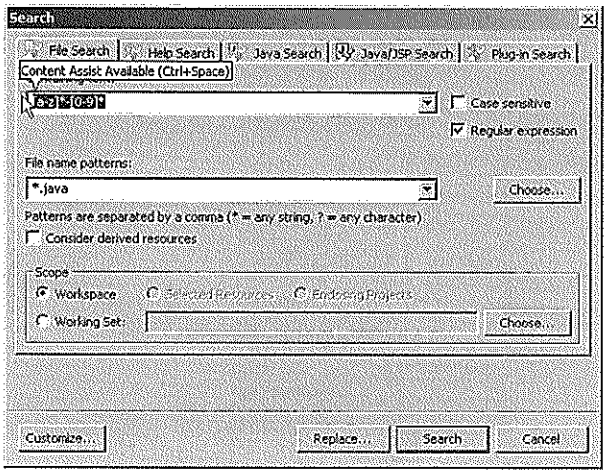


Figure 13-9. Recherche au moyen d'expressions régulières

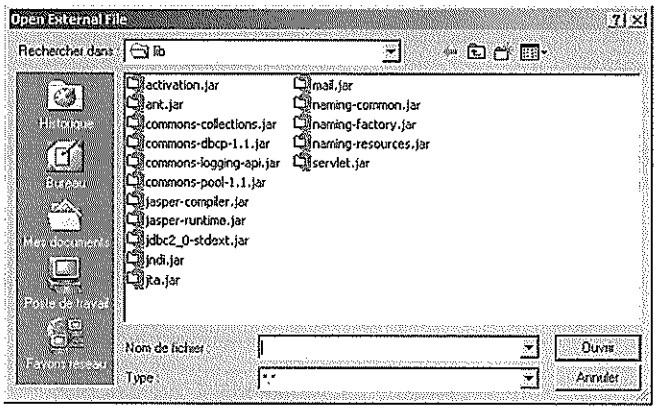


Figure 13-10. Ouverture d'un fichier externe

Window (Fenêtre) → Preferences (Préférences) → Workbench (Plan de travail) → Editors (Editeurs) de la version 3.0, il y en a maintenant vingt-trois, réparties sur trois pages – Annotations (Annotations), Quick Diff et Text Editor (Editeur de texte) –, comme on peut le voir dans la figure 13-11.

En marge de l'utilisation du soulignement par ligne droite ou ondulée en guise d'annotation, il est aussi possible d'afficher des annotations au moyen de surlignement en arrière-plan.
De nouvelles fonctions apparaissent aussi dans tous les éditeurs. On peut par exemple déplacer des lignes de code vers le haut ou vers le bas au moyen de Alt+FlècheHaut and Alt+FlècheBas, ou encore insérer une nouvelle ligne au-dessus ou au-dessous de la ligne courante grâce à Ctrl+Shift+Entrée et à Shift+Entrée. On peut convertir un texte sélectionné en minuscules ou en majuscules au moyen de Ctrl+Shift+Y ou de Ctrl+Shift+X. Enfin, on peut activer la barre Quick Diff au moyen de Ctrl+Shift+Q (ou

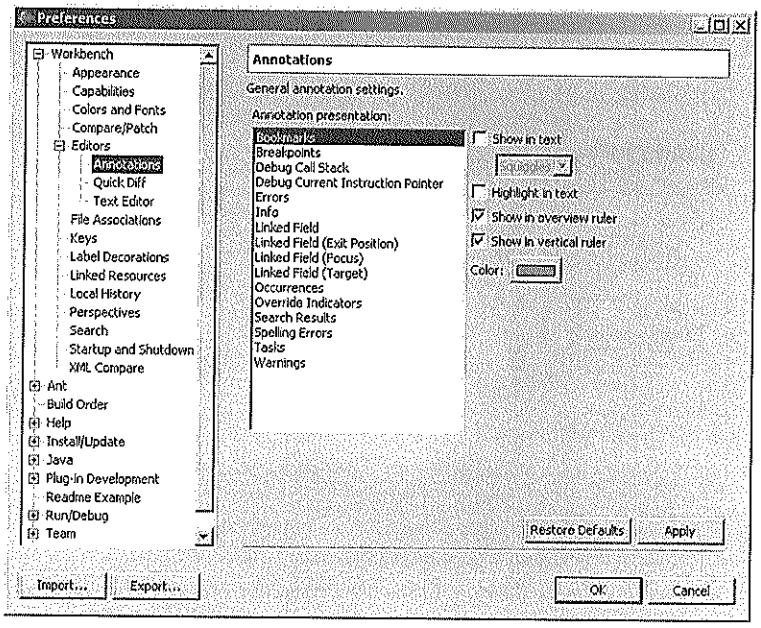


Figure 13-11. Personnalisation des annotations

encore grâce à la page Window (Fenêtre) → Preferences (Préférences) → Workbench (Plan de travail) → Editors (Editeurs) → Quick Diff).

Un objectif majeur d'Eclipse 3.0 est celui d'une meilleure ergonomie. De nombreuses boîtes de dialogue ont été repensées pour les rendre plus faciles à utiliser ; par exemple, la nouvelle boîte de dialogue Customize Perspective (Personnaliser la perspective), accessible au moyen de Window (Fenêtre) → Customize Perspective (Personnaliser la perspective), apparaît dans la figure 13-12. L'agencement général et en particulier les cases à cocher rendent celles-ci plus ergonomiques.

L'équipe de développement d'Eclipse travaille aussi sur la courbe d'apprentissage de cet EDI. Les conclusions du groupe de travail ne sont pas encore arrêtées, mais selon lui (citons ses membres) :

> Les utilisateurs novices d'un produit basé sur Eclipse peuvent vivre une première expérience désagréable, par surabondance d'informations. Cette première expérience pourrait être mieux vécue si le produit pouvait préconfigurer le plan de travail afin de ne proposer qu'un sous-ensemble de fonctionnalités réellement nécessaires et utilisables par un novice ; des pages de bienvenue pourraient être personnalisées pour des utilisateurs types ou des niveaux d'expérience particuliers.

L'équipe de développement d'Eclipse reconnaît clairement que la courbe d'apprentissage est trop raide. La version 3.0 propose plus de soutien à ses utilisateurs en fournissant plus d'aide sous la forme de liens actifs ou par l'intermédiaire de la touche F1.

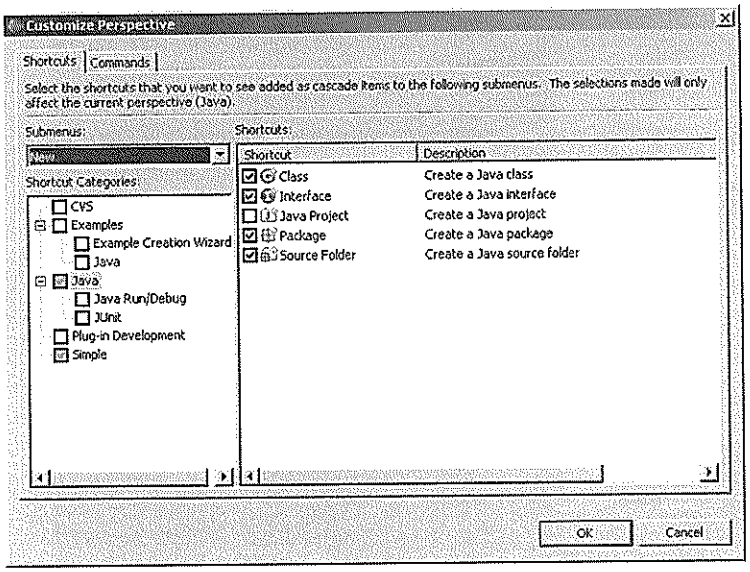


Figure 13-12. La nouvelle boîte de dialogue Customize Perspective (Personnaliser la perspective)

Beaucoup d'autres propositions de modification ont été formulées pour la plateforme Eclipse – en voici quelques-unes :

- Implémentation de la fragmentation de fenêtre dans les éditeurs de texte pour permettre à l'utilisateur de voir deux sections de texte en même temps.
- Stockage de la configuration et des préférences de l'utilisateur indépendamment de l'espace de travail afin de pouvoir les réutiliser dans d'autres espaces de travail.
- Possiblité d'ajout et de suppression de plug-in en dynamique, sans avoir à relancer Eclipse.
- Possibilité de désactiver un plug-in pour limiter la charge des ressources du système.
- Ajout d'un modèle de sécurité, incluant l'authentification de l'utilisateur, utilisable par tous les plug-in.
- Utilisation du support XML sous le J2SE 1.4 au lieu du plug-in Xerces.
- Portage de SWT sur des environnement 64 bits.

Modifications effectuées dans les outils de développement Java

Les améliorations majeures d'Eclipse 3.0 concernent le JDT, comme on pouvait l'espérer. Celles-ci sont principalement orientées vers plus d'ergonomie et de simplicité, et ont pour but d'aider le développeur Java à programmer plus vite et mieux. Il est difficile de dire lesquelles seront conservées dans la version destinée à la diffusion ; on peut néanmoins passer en revue celles qui sont actuellement implémentées dans la version bêta.

Vues Quick Hierarchy (vues rapides « Hiérarchie »)

On peut maintenant sélectionner un type, une méthode ou un paquetage dans l'éditeur du JDT et presser Ctrl+T pour obtenir une vue rapide de la hiérarchie des types (voir figure 13-13).

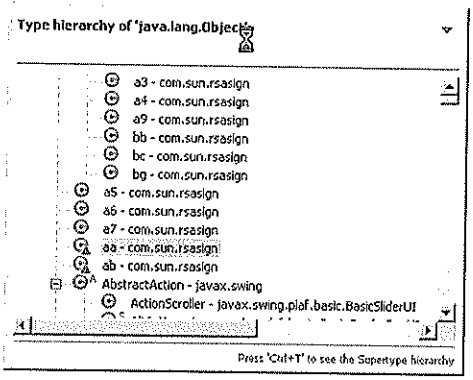


Figure 13-13. Visualisation rapide d'une hiérarchie

On peut aussi ouvrir une vue montrant la hiérarchie d'appel d'une méthode en sélectionnant Navigate (Naviguer) → Open Call Hierarchy (Ouvrir une hiérarchie d'appel) – ou en pressant Ctrl+Alt+H –, sous l'éditeur du JDT (ou, pour cet aspect, toute vue Java contenant des méthodes).

Création de constructeurs à partir de données membres

On peut créer un constructeur initialisant plus facilement les données membres d'une instance de classe grâce à Eclipse 3.0. Imaginons par exemple une classe contenant la données membre de type `String` :

```
public class Ch13_01 {

        private String text;

        public static void main(String[] args) {
                System.out.println("Bienvenue dans Eclipse 3.0");
        }

}
```

Pour créer un constructeur initialisant cette donnée membre, on peut maintenant sélectionner Source (Source) → Generate Constructor Using Fields (Génération de constructeur à partir des données membres), ce qui ouvre la boîte de dialogue de la figure 13-14.

On peut alors sélectionner la donnée membre à initialiser dans le constructeur et cliquer OK ; dans l'exemple présent, voici le constructeur ainsi créé :

```
public class Ch13_01 {
```

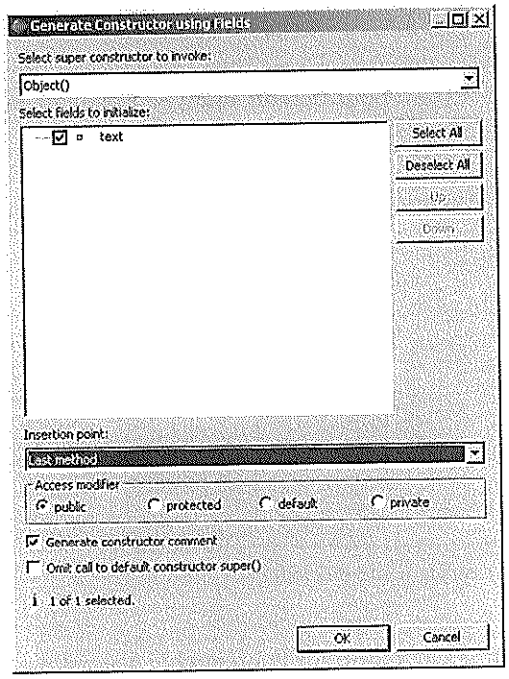


Figure 13-14. Création d'un constructeur initialisant une donnée membre

```
        private String text;

        public static void main(String[] args) {
                System.out.println("Bienvenue dans Eclipse 3.0");
        }

        /**
         * @param text
         */
        public Ch13_01(String text) {
                super();
                this.text = text;
        }
}
```

Création de méthodes modifiées

On peut aller plus loin – il est possible de convertir le constructeur ci-dessus en une méthode modifiée au moyen de la nouvelle option Refactor (Propagation des modifications) → Introduce Factory (Introduire une méthode). Il suffit pour cela de sélectionner la déclaration du constructeur ou un appel de celui-ci, sous l'éditeur du JDT, puis l'option de menu ci-dessus. Imaginons par exemple que le constructeur soit appelé dans la méthode main de la classe `Ch13_01` :

```
public class Ch13_01 {
```

```
        private String text;

        public static void main(String[] args) {
                Ch13_01 ch13_01 = new Ch13_01("Bienvenue");
                System.out.println("Bienvenue dans Eclipse 3.0");
        }

        /**
         * @param text
         */
        public Ch13_01(String text) {
                super();
                this.text = text;
        }
}
```

On sélectionne l'appel de ce constructeur puis l'option Refactor (Propagation des modifications) → Introduce Factory (Introduire une méthode), ce qui ouvre la boîte de dialogue de la figure 13-15.

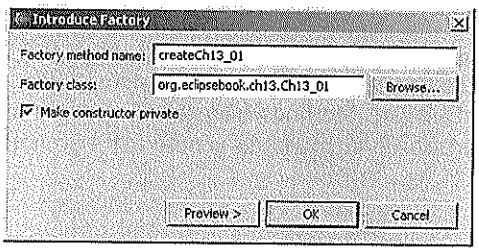


Figure 13-15. Création d'une méthode modifiée

On clique OK. Une nouvelle méthode modifiée est alors créée, `createCh13_01`, qui remplace l'appel du constructeur dans la méthode main, ce constructeur étant alors qualifié de private :

```
public class Ch13_01 {

        private String text;

        public static void main(String[] args) {
                Ch13_01 ch13_01 = createCh13_01("Bienvenue");
                System.out.println("Bienvenue dans Eclipse 3.0");
        }

        public static Ch13_01 createCh13_01(java.lang.String text) {
                return new Ch13_01(text);
        }

        /**
         * @param text
         */
        private Ch13_01(String text) {
                super();
```

```
            this.text = text;
        }
}
```

Par ailleurs, il est désormais possible de déplacer des données membres de type public static final entre les classes et les interfaces, de même que des types statiques internes entre classes. L'option Refactor (Propagation des modifications) → Change Method Signature (Changer la signature de la méthode) permet de plus de renommer les paramètres de méthodes surchargées.

Une tendance générale d'Eclipse 3.0 est de donner au programmeur plus de contrôle sur les tâches qu'il effectue déjà. Par exemple, la boîte de dialogue ouverte par la commande Source (Source) → Add Constructors from Superclass (Ajouter les constructeurs de la superclasse) de l'éditeur du JDT permet de sélectionner le constructeur de la superclasse qui doit être ajouté dans la classe courante. D'autres commandes donnent beaucoup plus de contrôle, comme par exemple les boîtes de dialogue permettant de générer du code à partir des commandes Source (Source) → Generate Getter and Setter (Générer les méthodes Getter et Setter), Source (Source) → Override/Implement Methods (Remplacer/Implémenter les méthodes) et Source (Source) → Generate Delegate Methods (Générer les méthodes déléguées). On peut indiquer le lieu d'insertion de la méthode générée et – pour les getter et les setter – l'ordre d'insertion.

À propos des méthodes getter/setter, il est bon de noter l'existence de nouveaux modèles pour celles-ci dans la page Window (Fenêtre) → Preferences (Préférences) → Java → Code Generation (Génération de code) → Code and Comments (Code et commentaires).

Mode d'insertion habile (smart insert mode)

L'éditeur du JDT propose un nouveau mode de saisie – *smart insert mode* – en plus des modes traditionnels en insertion directe et en écrasement. Le mode d'insertion habile (smart insert mode) propose des fonctionnalités supplémentaires au programmeur Java, telles que le complètement automatique de chaînes Java à des lignes de code en cours de frappe. On peut basculer entre ces trois modes en pressant la touche Insert, le mode courant étant indiqué dans la barre d'état (on est en mode Smart Insert dans la figure 13-8).

Pour configurer le mode d'insertion habile, il suffit de sélectionner Window (Fenêtre) → Preferences (Préférences) → Java → Editor (Editeur) → Typing (Frappe), ce qui ouvre la boîte de dialogue de la figure 13-16. En marge des options proposées par celle-ci, d'autres peuvent être envisagées, comme par exemple l'insertion automatique de points-virgules à la fin des lignes.

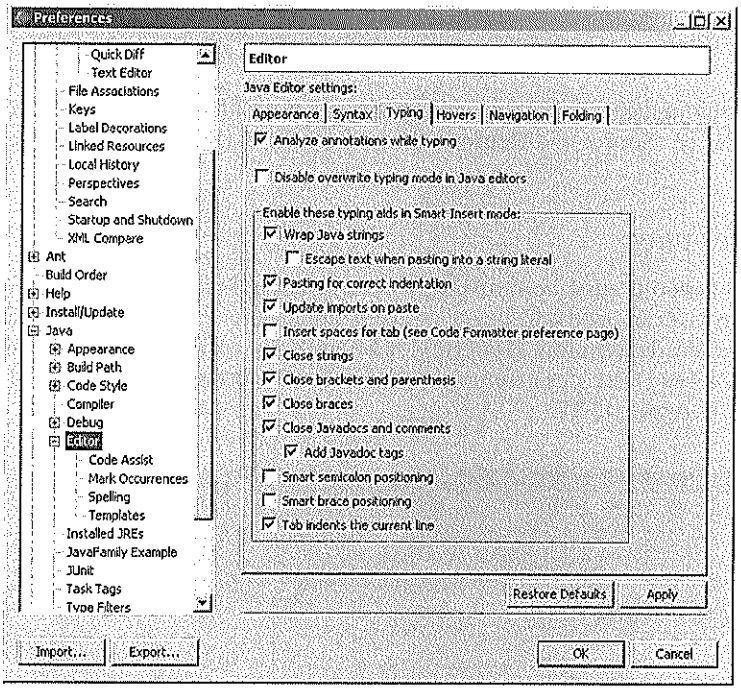


Figure 13-16. Configuration du mode d'insertion habile

Création de blocs de commentaires

En plus des lignes individuelles de commentaires venant décrire des blocs de code, il est possible d'insérer des blocs de commentaires Java. Par exemple, si l'on sélectionne le code suivant :

```
public class Ch13_01 {

        private String text;

        public static void main(String[] args) {
                System.out.println("Bienvenue dans Eclipse 3.0");
        }
}
```

et si l'on sélectionne l'option Source (Source) → Add Block Comment (Ajouter un bloc de commentaire) – on notera que l'option Source (Source) → Comment (Commentaire) est toujours disponible pour commenter du code au moyen d'une seule ligne –, on obtient pour résultat :

```
public class Ch13_01 {

        private String text;

/*        public static void main(String[] args) {
```

```
                System.out.println("Bienvenue dans Eclipse 3.0");
            }
*/}
```

Pour ne plus mettre un bloc en commentaire, il suffit de sélectionner celui-ci puis l'option Source (Source) → Remove Block Comment (Enlever un bloc de commentaire).

Nouvelles vues

Eclipse 3.0 propose aussi de nouvelles vues : Javadoc, Error log (Liste des erreurs), et Declaration (Déclaration). Celles-ci peuvent apparaître à l'écran comme toute autre vue, au moyen de Window (Fenêtre) → Show View (Afficher la vue). Par exemple, la vue Error log (Liste des erreurs), qui visualise les erreurs du fichier Eclipse *.log*, a l'allure de la figure 13-17.

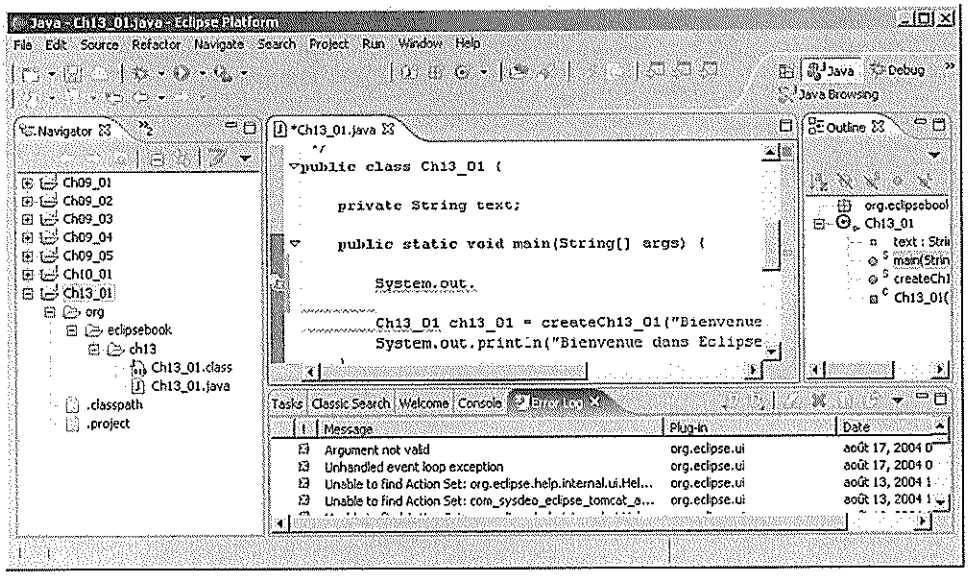


Figure 13-17. La vue Error log (Liste des erreurs)

Modifications complémentaires

Bien d'autres modifications, de moindre importance, ont été aménagées. Voici une liste non exhaustive des plus notables d'entre elles :

- Lorsqu'une variable locale est sélectionnée dans l'éditeur du JDT et lorsqu'on presse F3 – ou lorsqu'on sélectionne Navigate (Naviguer) → Open Declaration (Ouvrir une déclaration) –, l'éditeur du JDT affiche la déclaration de la variable.
- Le compilateur du JDT effectue plus de vérifications pour identifier et marquer des problèmes de style. Pour activer celles-ci, il suffit de sélectionner Window (Fenêtre) → Preferences (Préférences) → Java → Compiler (Compilateur) → Style (Style).

- Au lieu d'ajouter une accolade fermante dès qu'une accolade ouvrante est tapée, l'éditeur du JDT attend maintenant qu'une ligne entière de code soit tapée.
- L'éditeur du JDT propose de nouveaux assistants, les Assistants rapides – Split and join variable declaration (scission et regroupement de déclarations de variables), and Create field from parameter (Créer une donnée membre à partir d'un paramètre).
- La page de configuration de lancement permet désormais d'initialiser des variables d'environnement.
- Le compilateur du JDT peut maintenant indiquer si un projet est compilé alors qu'il inclut une bibliothèque binaire dont la version est incompatible avec le code généré. Cela peut se produire par exemple si l'on tente de générer des fichiers class compatibles avec la version 1.1 alors que la compilation utilise une bibliothèque Java 1.3. On se réfèrera à l'option Window (Fenêtre) → Preferences (Préférences) → Java → Compiler (Compilateur) → Build Path (Chemin de compilation).
- On peut aussi implémenter des recherches incluant des projets contenant le projet courant.
- Les boutons Go To Next/Previous Error (Atteindre l'erreur suivante/précédente) de la barre d'outils ont été remplacés par des boutons Go To Next/Previous Annotation (Atteindre l'annotation suivante/précédente) de type déroulant, qui permettent de choisir vers quelle annotation l'on veut se déplacer.
- Le compilateur du JDT peut trouver et marquer les exceptions non utilisées, qui sont déclarées mais pas levées. Cette option est activée au moyen de Window (Fenêtre) → Preferences (Préférences) → Java → Compiler (Compilateur) → Unused Code (Code non utilisé).

De nombreuses pistes d'amélioration sont envisagées, qui seront peut-être incluses dans la version finale d'Eclipse 3.0. En voici quelques-unes :

- Extension de l'éditeur du JDT pour qu'il puisse gérer des fichiers sources de type Java, comme par exmple le SQLj et le JSP.
- Marquage de méthodes surchargées au moyen d'indicateurs de surcharge.
- Gestion des nouvelles fonctionnalités et des extensions du J2SE 1.5, telles que les types génériques, l'autoboxing, les imports statiques, etc. La version finale d'Eclipse 3.0 sera probablement diffusée avant la version 1.5 du J2SE, mais l'équipe de développement d'Eclipse travaille d'ores et déjà sur l'intégration de cette version dans Eclipse 3.0.

Autres modifications

D'autres changements sont en cours dans d'autres sous-projets d'Eclipse 3.0, en particulier dans CVS, Ant, le PDE et le SWT.

Eclipse est désormais diffusé avec la dernière version de Ant (version 1.5.4). La configuration de lancement de Ant est entièrement nouvelle ; elle permet de travailler avec Ant sous une machine virtuelle Java séparée (le choix du JRE Ant étant à préciser dans l'onglet JRE de la configuration de lancement Ant).

En ce qui concerne CVS, on peut maintenant utiliser l'option Team (Equipe) → Show Annotation (Afficher les annotations) pour ouvrir la vue CVS Annotation (Annotation CVS). Si l'on sélectionne une ligne dans l'éditeur du JDT, la vue CVS Annotation (Annotation CVS) indiquera qui a produit la modification – ce qui signifie que l'on pourra tracer l'origine d'un bug.

Quelques améliorations ont aussi été aménagées dans le SWT. SWT accepte que des widgets Swing ou Ant soient intégrés dans SWT pour d'autres plateformes que Windows. À ce jour, cette possibilité n'est opérationnelle sous Windows que pour le JDK 1.4 et sous GTK et Motif que pour les premières versions du JDK 1.5 de Sun.

Les shells SWT peuvent aussi avoir des formes irrégulières. Ces formes peuvent être définies comme des combinaisons de rectangles et de polygones. SWT fournit aussi un nouveau widget explorateur qui peut afficher des documents HTML. À ce jour, cependant, ce widget n'est supporté que sous Windows (il utilise l'Explorateur Internet version 5.0 ou ultérieure) et sous Linux GTK (il utilise Mozilla 1.4 GTK2).

On compte peu de modifications dans la version bêta actuelle d'Eclipse 3.0. Les deux plus notables sont que le PDE supporte maintenant les tests JUnit pour le développement de plug-in et qu'il existe un éditeur spécial de configuration de compilation permettant d'éditer le fichier *build.properties* d'un plug-in.

D'autres changements envisagés incluent l'amélioration des éditeurs et des débogueurs du PDE, ainsi que l'implémentation d'une aide contextuelle pour les plug-in.

Ainsi se termine la présentation succincte de l'état de l'art concernant la version 3.0 d'Eclipse. Bien entendu, les caractéristiques évoquées peuvent encore évoluer d'ici à la diffusion officielle de celle-ci.

Index

Symboles

& (esperluète), 172
* (astérisque), 19
@ (symbole arobas), 48
$ (symbole dollar), 61
″ (guillemets), 18
. (point), 19, 31
+ (signe plus), 109, 186
; (point-virgule), 18, 296
{} (accolades), 59, 299
// (commentaire sur une seule ligne), 36

A

about.html, 245
Abstracting Windowing Toolkit (voir AWT)
accolades {}, 59, 299
élément <action>, 255, 259
actions
  configuration pour une vue, 276
  contrôleurs et, 222
  création, 259-260
  définition, 255
  menus et, 258
  MultiPageEditorContributor.java, 269
<actionSet>, 255
AIX Motif, 150, 151, 157
Alt+FlècheBas, 290
Alt+FlècheHaut, 290
Ant outil de
  fonctionnalité, 115-118
Apache Ant (voir outil de compilation Ant)
appels de méthode
  exécution du code en pas à pas, 79
  hiérarchie de, 293
  mise à jour, 49
  récursif, 75
<APPLET> balise HTML, 132
applets, 131-134
Applets Java, 131-134
applications
  basées sur le web, 216-219
  Eclipse 3.0 et, 288
  test au moyen de JUnit, 65
  test d'application avec JUnit, 68-75
arbres généalogiques de classes, 13
arbres, SWT et, 184-186
architecture de type Modèle 1, 221
architecture de type Modèle 2, 221
architecture Modèle-Vue-Contrôleur (MVC), 221
architecture MVC (Modèle-Vue-Contrôleur), 221
arguments
  outil de compilation Ant, 125
  ligne de commande, 45
  méthodes et, 19
arguments en ligne de commande, 45
assistant au codage
  Editeur Ant et, 125
assistant de code
  activation, 19
  création de nouvelles options, 61
  généralités, 17-19
  Javadoc et, 48
  ponctuation et, 19, 19, 31
  raccourci, 30, 48, 61
assistant du PDE, 278

assistant JUnit, 68, 71
assistant Nouvelle extension, 256
assistants
  assistant Nouvelle extension, 256
  création d'éditeurs multipage, 265-273
  création de plug-in, 247
  création de vues, 273-279
  PDE et, 247
Assistants rapides (éditeur du JDT), 299
astérisque (*), 19
attribut action (formulaires), 225
attribut depends, 119
attribut METHOD (formulaire HTML), 202
attribution d'une licence, 4
attributs, assistant au codage et, 125
authentification de l'utilisateur, 292
@author (Javadoc), 48
autoboxing, 299
Avancer d'un pas avec entrée (F5), 79, 82
Avancer d'un pas avec filtres (Shift+F5), 79
Avancer d'un pas sans entrée (F6), 80
AWT (Abstract Windowing Toolkit)
  Eclipse 3.0, 300
  GUI et, 134-136
  SWT et, 149

B

balises <%...%>, 199
barre d'outils
  SWT et, 176-180
barre d'outils (classe SWT), 176
barre de défilement
  arbres et, 184, 184
  boîtes de dialogue et, 188
  curseurs et, 181, 181, 183
barre de marquage, 22
barres d'outils
  actions et, 255
  Eclipse 3.0 et, 288
  jeux d'actions et, 256
  MultiPageEditorContributor.java, 269
  plan de travail Eclipse et, 7
  verrouillage, 58
barres de défilement
  styles de liste, 165
bean formulaire, 237
bibliothèques, création de projets et, 13
bien écrit (XML), 127
blocs de commentaires, 297, 298
boîte de dialogue A propos, 176
boîte de dialogue A propos des plug-in Plateforme Eclipse, 246
boîte de dialogue Ajout au contrôle de version CVS, 99
boîte de dialogue Ajout d'un référentiel CVS, 97
boîte de dialogue Changement de nom de la méthode, 49
boîte de dialogue Changement du nom du type, 50
boîte de dialogue Comparer avec l'historique local, 103
boîte de dialogue Configuration de la connexion de la source, 68
boîte de dialogue Création d'une nouvelle branche CVS, 113
boîte de dialogue Customize Perspective (Personnaliser la perspective), 291
Boîte de dialogue de confirmation d'effacement de projet, 25
Boîte de dialogue de saisie de texte, 144
boîte de dialogue Exécuter, 46, 132, 157
boîte de dialogue Exporter, 280
boîte de dialogue Extraction d'interface, 53
boîte de dialogue Fusionner, 114
boîte de dialogue Génération d'un Javadoc, 48
boîte de dialogue Générer les méthodes Getter et Setter, 37
boîte de dialogue Importation, 26
boîte de dialogue New Java Class (Nouvelle classe Java)
  création de classes, 285
boîte de dialogue New project (Nouveau Projet)
  création de projets, 284
boîte de dialogue Nouveau, 68
boîte de dialogue Nouveau dossier, 204, 206
boîte de dialogue Nouveau jeu de documents, 61
boîte de dialogue Nouveau Projet
  ajout de fichier JAR dans le classpath, 43
  création d'éditeurs multipage, 265
  création de projets, 13
  créationg de plug-in, 247
  créer un projet, 143
  Développement de plug-in et Projet de plug-in, 254
  sélection de projets, 211
boîte de dialogue Nouveau projet Java

création de projets, 39, 118
création de servlets, 204
boîte de dialogue Nouveau projet Tomcat, 211
boîte de dialogue Nouvelle classe Java
"Type de déclaration", 34
création de classes, 15
plusieurs classes, 33
Boîte de dialogue Nouvelle entrée de variable, 67, 68
boîte de dialogue Nouvelle entrée du chemin de variables, 68
boîte de dialogue Nouvelle page de testeur de code, 20
boîte de dialogue Partager le projet avec le référentiel CVS, 97
boîte de dialogue Personnalisation de la perspective, 61
boîte de dialogue Préférences, 124
activation de l'assistant de code, 19
ajout de variables de classpath, 68
filtres de progression, 80
installation de JUnit, 67
option de menu Ant -> Editeur, 124
personnalisation de la génération de code, 59
répertoire d'installation de Tomcat, 210
boîte de dialogue Principaux paramètres de la vue, 273
boîte de dialogue Propriétés
ajout d'un fichier JAR, 42
ajout d'un projet dans le chemin de la génération, 39
ajout de fichiers JAR, 42
bouton Ajouter un dossier classe, 44
onglet Ordre et exportation, 44
variables de classpath et, 44
boîte de dialogue Propriétés de point d'arrêt de ligne Java, 82
boîte de dialogue Propriétés de point d'arrêt de méthode Java, 86
boîte de dialogue Propriétés de point de contrôle Java, 86
boîte de dialogue Propriétés de points d'arrêt, 84
boîte de dialogue Recherche, 56, 57, 57
boîte de dialogue Sélection d'un jeu de documents, 61
boîte de dialogue Sélection du dossier, 204, 223
boîte de dialogue Sélectionner un jeu de documents, 61
Boîte de dialogue V4E, 144
boîte de dialogue Valider, 104
boîtes de dialogue, 187-192
boîtes de texte, 238
bordures, caractéristiques, 178
Borland, 2
boucles infinies, 78
bouton radio, 177
bouton Supprime tous les programmes terminés (Débogage), 212
boutons
ajout, 158-162
boîtes de dialogue et, 188, 191
contrôles comme des, 152
Eclipse 3.0 et, 286, 288
jeux d'actions et, 255, 256
plug-in Easy Struts, 238
points d'extension et, 247
radio, 177
branches
création, 112
fusion, 114
nœud de branches, 99
support CVS, 95
versions du code et, 93
bulles d'aide
assistant de code et, 18
Editeur Ant, 127
Quick Fix et, 23
Button (contrôle SWT), 152

C

cadre de développement Struts
création de contrôleurs, 228-231
création de modèles, 231-236
création de vues, 225-227
Eclipse et, 222-225
généralités, 221
nettoyage d'applications, 221
plug-in Easy Struts, 236-243
Canvas (contrôle SWT), 152
Caret (contrôle SWT), 152
case à cocher, 177
case à cocher Fenêtre -> Préférences -> Plan de travail -> Effectuer une compilation automatique lorsqu'une ressource est modifiée, 90
cases à cocher, 238

champ texte
  boîtes de dialogue, 191
  curseurs, 181
  exemple de menu, 176
champs
  types des données de, 19
checkmarks, 186
chemin de classe
  projets et, 13
chemin de compilation
  fichier servlet.jar, 205
  projets et, 39, 39
  SWT et, 150, 151
chemin de génération
  fichier servlet.jar, 202, 224
choix de menu Afficher, 21
choix de menu Définit les importations, 21
choix de menu Fenêtre -> Afficher la vue, 9
fichiers .class, 40-45
classe Action, 255
classe AppFrame, 134, 135
classe Applet, 131
classe AWT Frame, 134
classe Button (SWT), 158, 158
classe DirectoryDialog (SWT), 187
classe FileDialog (SWT), 187
classe Frame (AWT), 134
classe HttpServlet, 42, 202
classe java.lang.Object, 15
classe javax.servlet.http.HttpServlet, 202
classe JFrame (Swing), 137, 139
classe JPanel (Swing), 136
classe Listener, 178
classe MessageDialog, 261
classe MultiPageEditor, 269
classe MultiPageEditorPart, 269
classe org.eclipse.ui.javaPerspective, 262
classe org.eclipse.ui.resourcePerspective, 263
classe Panel (Swing), 137
classe SampleNewWizard, 267, 268, 269
classe SampleNewWizardPage, 268
classe SampleView, 276
classe StringTokenizer, 271
classe Swing JFrame, 137, 139
classe Swing JPanel, 136
classe SWT barre d'outils, 176
classe SWT DirectoryDialog, 187
classe SWT FileDialog, 187
classe SWT FillLayout, 163
classe SWT FormLayout, 163
classe SWT GridLayout, 163
classe SWT List, 165, 165
classe SWT RowLayout, 163
classe Text (SWT), 158, 158
classe ToolItem, 177
classe Tree (SWT), 184
classe TreeItem, 184, 185, 186
classe UIManager, 139, 139
classe ViewContentProvider, 277
classe ViewPart, 276
classe WindowAdapter, 137
classe Wizard, 267
classement (tri), 271
classes
  accès à tous les membres, 19
  anonyme, 54
  anonymes, 214
  basé sur JUnit, 65
  basées sur JUnit, 69
  création, 15, 17, 31-38, 285
  données membres static final et, 296
  imbriquées, 54
  importation, 38
  ordre des chemins de classe, 13
  plug-in et, 247
  recherche de tous les membres, 17
  renommage, 50, 50
  sous forme de servlets, 203
  stockage d'une nouvelle, 213
classes anonymes, 54, 214
classes imbriquées, 54
classes Java (voir classes)
classpath, 42, 42, 44
clic de bouton droit, 278
code (voir aussi Java)
  actions et, 259
  assistant de code et, 17
  branches et, 112
  comparaison avec l'historique, 103
  compilation automatique, 90
  contrôle du code source, 93, 93
  débogage, 77, 78, 79, 90
  éditeurs multipage, 265-267, 269-273
  édition, 9, 146-147
  espace de travail et, 8
  génération, 36, 59, 146-147
  mise à jour, 101-103
  modification de code durant une session de débogage, 90-92

pages de testeur de code et, 20
personnalisation, 59
pour des plug-in, 260-261
projets Java, 11-14
recherche, 56-57
référentiels pour, 95
répertoires pour, 224
synchronisation, 105-108
validation, 102, 103, 105
vue Packages, 19
code source (voir code)
Combo (contrôle SWT), 152
commande CVS, 95
commande OLE.OLEIVERB_INPLACEACTIVATE, 193
commande set, 198
commande setenv, 198
commande shutdown (Windows), 198
commande shutdown.sh (Unix), 198
commande startup (Windows), 198
commande startup.sh (Unix), 198
commentaires
création de blocs de commentaires, 297, 298
personnalisation de la génération de code, 59
sur une seule ligne (//), 36
symbole arrobas et, 48
Common Public License (voir CPL)
compilation automatique, 90
Composite (contrôle SWT), 152
composites, 152, 163-164
compteur d'occurrences, 81-84, 86
Concurrent Versions System (voir CVS)
conditions
configuration, 84-85
points d'arrêt sur une exception et, 87
configuration
serveur d'applications Tomcat, 197-199
actions de la vue, 276
configurations d'exécution, 157
mode d'insertion habile, 296
outil de compilation Ant, 124-127
plug-in, 248
points d'arrêt, 84-87
configuration d'exécution, 45-46, 157, 168
conflits
CVS et, 102
validation de code et, 103
console
Ant et, 117, 128
compiler les projets, 122
constante SWT.ARROW_DOWN, 181
constante SWT.ARROW_UP, 181
constante SWT.DRAG, 181
constante SWT.END, 181
constante SWT.HOME, 181
constante SWT.NONE, 177
constante SWT.PAGE_DOWN, 181
constante SWT.PAGE_UP, 181
constructeurs, 36, 57, 293-294
contexte d'exécution empilés, 75
contextes d'exécution empilés, 82
contrôle du code source, 93
contrôle texte
barre d'outils et, 178
boutons et, 158
Classe Text, 158
glissières, 181
styles pour, 159
contrôles (voir aussi texte)
ajout de boutons, 158-162
arbres comme, 184-186
Barre d'outils comme, 176-180
boîtes de dialogue comme, 187-192
classe SampleNewWizardPage, 268
composites et gestionnaire de placement, 163-164
curseurs comme, 180-184
Easy Struts et, 238
fixer les coordonnées, 161
formulaires Struts et, 225
listes et, 165-167
options SWT, 152-155
positionnement, 163
widgets et, 150, 152
contrôleurs
formulaires Struts et, 226
menus comme, 171-176
conversion en majuscules, 290
conversion en minuscules, 290
conversion majuscule/minuscule, 290
CoolBar (contrôle SWT), 152
CoolItem (contrôle SWT), 152
coordonnées, paramètrage des, 161
copyleft, 4
correctifs, création, 108, 110
CPL (Common Public License), 4
Ctrl+Espace (assistant de code)

description, 30, 61
paramètres de méthode, 48
Ctrl+Espace (assitant au codage)
Atributs des éléments syntaxiques de Ant, 125
Ctrl+Shift+Entrée, 290
Ctrl+Shift+Q, 290
Ctrl+Shift+X, 290
Ctrl+Shift+Y, 290
Ctrl+Space (assistant de code)
comme raccourci, 19
curseurs, 180-184
CVS (Concurrent Versions System)
connexion à, 95
Eclipse 3.0 et, 300
emplacement des données administratives, 99
modules dans, 94, 98
comme open source, 93
support d'équipe comme, 8
CVSNT, 95, 95

D

débogage
arrêt, 78
compteur d'occurrences de point d'arrêt, 81-84
Eclipse 3.0 et, 300
evaluation d'expressions, 89
évaluation d'une expression, 89
exécution du code en pas à pas, 79
exemple, 75, 76
généralités, 75
JSP, 216
modification de code durant une session de débogage, 90-92
multi-threads, 85
perspectives et, 10
points d'arrêt et, 76-79, 84-87
projets pour le Web, 215
projets web, 216
reprise de l'exécution, 81
déclarations
fichiers JSP et, 199
ouverture, 55
recherche de, 57
variables locales, 298
décodage de fichier, 206
dépendances de génération, 234
dépendances, génération, 235
@deprecated (Javadoc), 48
développement pour le web
connexion aux JavaBeans, 207-209
création de contrôleurs, 228-231
création de JSP, 199-201
création de modèles, 231-236
création de servlets, 201-204
création de servlets avec gestion automatique des répertoires, 204-207
création de vues, 225-227
débogage et, 215, 216
déploiement d'applications, 216-219
plug-in Easy Struts, 236-243
plug-in Tomcat Sysdeo, 209-216
serveur d'applications Tomcat et, 197-199
Struts et Eclipse, 222-225
extension .do, 228, 229
documents web, 200
données membres
constructeurs basés sur, 293-294
Eclipse 3.0, 296
dossier bin, 118
dossier src
contenus, 118
dossiers
affecter les propriétés, 119
modules et, 94
projets et, 8, 13
double-clic, 279

E

Eclipse 3.0
changement de sous-projet, 299
création de projets Java, 284-288
description, 283
JDT et, 292-299
modifications dans la plateforme, 288-292
modifications des sous-projets, 300
EDI (environnement de développement intégré), 1, 245
EDI Java (voir JDT Eclipse)
éditeur de fichier manifeste de plug-in, 248, 249, 251
éditeur du JDT
accolade fermante et, 299
Assistants rapides, 299

commande Source (Source)–> Add Constructor from Superclass (Ajouter un constructeur de la superclasse), 296
hiérarchie d'appel de méthode, 293
mode d'insertion habile, 297
remplacements de boutons, 299
variables locales et, 298
vue CVS Annotation (Annotation CVS) et, 300
éditeur JDT
barre Quick Diff, 286
erreurs de syntaxe, 127
instauration d'un point d'arrêt, 76
mode d'insertion habile (smart insert mode), 296
Ouvrir la hiérarchie des types, 55
Ouvrir une déclaration, 55
smart insert mode (mode d'insertion habile), 286
vue Recherche, 57
éditeur V4ALL, 143
éditeur visuel de V4E, 143
éditeurs (voir aussi éditeur JDT)
Eclipse 3.0 et, 292
éditeur V4ALL, 143
élaboration d'annotations, 289
fichier manifeste de plug-in, 248, 249, 251
fichiers et, 32, 289
fonctions d'édition, 290
manipulation de fenêtre, 9
multipage, 265-273
outil de compilation Ant, 117
perspective Débogage et, 79
perspective Java, 13
plan de travail et, 9
vue Packages, 19
XML Buddy, 200
éditeurs de texte (voir éditeurs)
éditeurs multipage, 265-273
éditeurs XML, 124, 200
élément de type form beans, 222
élément getProperty (JSP), 208
élément jsp:getProperty, 208
élément jsp:setProperty, 208
élément jsp:useBean, 208
élément servlet (XML), 203, 206, 228
élément servlet-mapping (XML)
contrôleurs, 228
regroupement, 206
Tomcat et, 203, 214
élément setProperty (JSP), 208
élément TOMCAT_HOME, 237
élément UI (SWT), 152
élément useBean (JSP), 208
éléments
déplacement, 52, 52
recherche de, 57
renommage, 48-51
éléments syntaxiques
assistant au codage et, 125
empilement de piles, 78
enfants, nœuds d'arbre, 185
environnement de développement de plug-in (voir PDE)
environnement de développement intégré (EDI), 1, 245
environnement GTK2 Linux, 7
environnement Linux
CVS et, 93, 95
OLE et, 192
raccourcis Eclipse, 5
SWT et, 150
environnement Linux GTK
configuration d'exécution, 157
SWT et, 150, 151, 300
widgets explorateurs et, 300
environnement Linux Motif
configuration d'exécution, 157
SWT et, 150, 151, 300
environnement Mac, 149
environnement Motif Linux
portage d'Eclipse, 7
environnement Unix
CVS et, 93, 95
raccourcis Eclipse, 5
serveur Tomcat web, 198
environnement Win32, 151, 157
environnement Windows
AWT et, 149
CVS and, 95
CVS et, 93
look-and-feel, 139, 139, 141
OLE et, 9, 192
portage d'Eclipse, 7
raccourci Alt+F, 172
raccourcis Eclipse, 5
serveur Tomcat web, 198
SWT et, 150

widgets explorateurs et, 300
erreurs dans les fichiers de compilation, 127-128
espace de travail
Eclipse 3.0, 292
édition de fichiers hors de, 289
fichiers correctifs et, 109
généralités, 8
projets sous forme de dossiers dans, 13
espace disque, libération, 288
espaces (blancs), 111
esperluète (&), 172
étiquettes (voir texte d'une étiquette)
comme des contrôles, 152
événement ControlListener (SWT), 160
événement fileExitItemListener, 173
événement fileSaveItemListener, 173
événement FocusListener (SWT), 160
événement KeyListener (SWT), 160
événement MouseListener (SWT), 160
événement MouseMoveListener (SWT), 160
événement MouseTrackListener (SWT), 160
événement SelectionListener (SWT), 160
événement SWT.Selection, 178
événements (voir récepteurs d'événements aussi)
curseurs et, 181
item de barre d'outils et, 178
UI thread et, 162
@exception (Javadoc), 48
exception java.lang.NullPointerException, 87
exception SWTException, 162
exceptions
compilateur du JDT et, 299
Javadoc et, 48
points d'arrêt et, 85, 87-87
Exécuter -> Ajouter un point d'arrêt d'exception Java, 87
Exécuter -> Ajouter/Supprimer un point de contrôle, 86
Exécuter -> Exécuter en tant que -> Plan de travail d'exécution, 252
Exécuter jusqu'à l'instruction de retour (F7), 80
exécution
points d'arrêt et, 86, 87
reprise en cours de débogage, 81
exécution ligne après ligne, 79
explorateur d'objets, 55
explorateurs
envoi de texte à, 199
explorateurs d'objets, 55
répertoire webapps et, 198
serveur web Tomcat et, 200
widgets et, 300
exportation
préférences, 64
projets, 13
tri d'éléments, 44
expressions
débogage et, 10, 89, 89, 89
Eclipse 3.0 et, 289
fichiers JSP et, 199
vue Expressions et, 79
expressions régulières (voir expressions)

F

F3 – Navigate (Naviguer)-> Open Declaration (Ouvrir une déclaration)—, 298
F5 (Avancer d'un pas avec entrée), 79, 82
F6 (Avancer d'un pas sans entrée), 80
F7 (Exécuter jusqu'à l'instruction de retour), 80
Fenêtre -> Préférences
assistant de code et, 19, 61
filtres de progression, 80
JUnit et, 67
option Variables de chemin d'accès aux classes, 44
personnalisation de la génération de code, 59
plug-in Easy Struts, 236
plug-in Tomcat, 209
selection du JRE, 46
fenêtres, 9
boîte de dialogue, 187
boîtes de dialogue, 191
exemple d'arbre, 186
objet barre de menu, 173
shells et, 152
feuille à dessin, 143, 144, 146
fichier ApplicationResources.properties, 231, 231, 235
fichier build.properties, 300
fichier build.xml
Ant et, 116, 117, 218
projets et, 119, 121

fichier descriptif de la bibliothèque de balises, 227
fichiers JAR (Archive Java)
accessibilité du code et, 19
création de projets et, 13
généralités, 40-45
localisation de, 68
outil de compilation Ant et, 118-123
plug-in et, 245, 247
plug-ins et, 280
répertoire lib et, 245
Struts et, 224, 236, 237
SWT et, 151
fichier junit.jar, 67
fichier plugin.properties, 245
fichier plugin.xml
élément <action>, 259
élément <actionSet>, 255
code XML, 254
création, 246
édition, 248, 251
fichier plugin.properties, 245
points d'extension, 262
répertoire des plug-in et, 245
fichier server.xml, 212, 212, 214
fichier servlet.jar
ajout dans le classpath, 42, 43
chemin de compilation et, 205, 224
création de servlets, 202
variables de classpath et, 44
fichier struts-config.xml, 229
fichier struts.jar, 224
fichier swt.jar, 150, 167
fichier web.xml
création locale, 214, 215
déploiement d'applications pour le web, 217
description, 203
édition, 205
exemple d'application Struts, 228
servlets et, 214
fichier XML, 124
fichiers
ajout dans un projet, 268
édition hors de l'espace de travail, 289
fichiers de compilation, 127-128
propagation des modifications, 52
recherche dans, 57
référentiel CVS et, 94
sauvegarde, 32
synchronisation, 98
tri, 206
validation, 99, 99
fichiers de compilation, 127-128
fichiers JSP, 299
fichiers manifestes de plug-in (voir fichier plugin.xml)
fichiers SQLj, 299
fichiers WAR (Web Archive), 216, 218
fichiers ZIP, 280
FillLayout (classe SWT), 163
filtres, 79, 107
filtres de progression, 80
Flash, 131
format GIF, 245
FormLayout (classe SWT), 163
formulaires
affichage de contrôles, 225
Struts et, 222
fragmentation, éditeurs de texte et, 292
fragments de plug-in, 6
fusion de branches, 114

G

gestionnaire de placement
structure, 163-164
gestionnaire des fenêtres, 152
gestionnaires de placement
SWT et, 158
glisser-déposer GUI, 142
GridLayout (classe SWT), 163
Group (contrôle SWT), 152
GUI (interface graphique utilisateur)
Applets Java et, 131-134
Applications Swing, 136-142
AWT et, 134-136, 149
fenêtre SWT et, 192
glisser-déposer, 142
module V4ALL, 143-148
modules et, 142
plan de travail Eclipse et, 7
plug-in et, 143
vues et, 9
guillements ("), 18

H

Hewlett Packard, 3
hiérarchies de type, 55
historique
comparaison de code avec, 103

modifications de ressource et, 8
Hitachi, 3
HPUX Motif
configuration d'exécution, 157
SWT et, 150, 151
HTML
attribut METHOD, 202
AWT et, 134
Java et, 201
pages JSP et, 238
serveur d'aide et, 9
widgets explorateurs et, 300
balise <html:form> (Struts), 225
balise <html:multibox> (Struts), 226
balise <html:options> (Struts), 226

I

IBM
Eclipse et, 2
OTI et, 3
Standard Widget Toolkit, 138, 149
images, barre d'outils et, 177
importation
aide AWT, 134
classes, 38
pages de testeur de code, 21
projets, 26
tri d'éléments, 44
index (curseur), 180
installation
serveur d'applications Tomcat, 197-199
Eclipse, 4, 5
JUnit, 67, 68
plug-ins, 6
problèmes, 5
instanciation d'objets, 208
Intel, 3
interface graphique utilisateur (voir GUI)
interface IStructuredContentProvider, 277
interface IStructuredSelection, 277
interface IWorkbenchWindow, 260
interfaces
données membres static final et, 296
extraction, 53-54
implementation, 33
Internet Explorer, 192-195, 203
Internet, applets Java et, 131
Item Action Performed (propriétés Evénements), 146

J

J2SE, 292, 299
Java
AWT et, 149
configuration d'exécution, 45-46
création de classes, 31-38
création de paquetages, 38-39
Eclipse 3.0 et, 286
Eclipse et, 1-6
fenêtre d'édition, 9
fichiers JAR et .class, 40-45
HTML et, 201
Javadoc et, 48, 48, 68
JRE et, 46
méthodes de création, 29-31
personnalisation de l'environnement pour, 58-64
perspectives et, 10
propagation des modifications, 48-54
recherche de code, 56-57
Swing et, 149, 150
SWT et, 150
variables d'environnement et, 198
visualisation de hiérarchies de type, 55
Java Development Toolkit (voir JDT)
fichier .java, 232
Java Native Interface (JNI), 150, 156
Java Runtime Environment (voir JRE)
Java Web Start, 131
java.applet.Applet, 131
java.lang.ArrayIndexOutOfBoundsException, 46
JavaBeans
architecture MVC et, 221
connexion à, 207-209
exemple d'application Struts, 232
méthodes Getter/Setter, 37
Javadoc, 48, 48, 68
JavaServer Pages (voir JSP)
JDK (Java Development Kit), 5, 300
JDT (Java Development Toolkit)
compilateur du JDT, 298
compilateur JDT, 119
débogueur JDT, 90
Eclipse 3.0 et, 292-299
Eclipse et, 3, 6, 11
éditeur pour, 9
Quick Fix et, 22-25
jeu d'actions
définition, 255

jeux d'actions
création, 256, 258
regroupement de plusieurs éléments, 258
jeux de documents, 57, 61
JFace, 7
JNI (Java Native Interface), 150, 156
jokers, 57
JRE (Java Runtime Environment)
Eclipse et, 5
modification de code durant une session de débogage et, 90
modification de la configuration par défaut, 91
plan de travail d'exécution, 252
sélection, 46
JSP (JavaServer Pages)
architecture MVC, 221
création, 199-201
débogage, 216
JavaBeans et, 207-209
plug-in Easy Struts et, 237
plug-in Tomcat Sysdeo et, 211, 212
servlets et, 201
JUnit, test au moyen de, 65-75, 300

L

Label (contrôle SWT), 152
langages, changement, 6
liens symboliques, 5
lignes, manipulation, 290
List (classe SWT), 165
List (contrôle SWT), 152
List classe (SWT), 165
listes, 152, 165-167
balise <logic:iterate> (Struts), 226
look-and-feel
choix, 139, 139
Plateforme Sun, 141
Swing, 150

M

Machine Virtuelle Java, 46, 156
marque HEAD, 95
membres
édition de valeurs en cours de débogage, 89
points d'arrêt et, 85
recherche de, 57
variables locales et, 54
Menu (contrôle SWT), 152
menu déroulant, 180
élément <menu>, 255
Menu objet, 171
MenuItem (contrôle SWT), 152
menus
actions et, 255, 258
création, 258, 258
Eclipse 3.0 et, 288
jeux d'actions et, 256
MultiPageEditorContributor.java, 269
plan de travail et, 7
points d'extension et, 247
structure, 171-176
SWT et, 180
MERANT, 2
Metal (look-and-feel), 139, 139, 141
méthod paint (Graphics), 131
méthode actionPerformed (UIManager), 139
méthode add (List), 165
méthode addListener, 178
méthode addPage (MultiPageEditorPart), 269
méthode addSelectionListener, 160
méthode allocate
description, 71
objectif, 67, 72
test de bon fonctionnement, 74
méthode assertEquals (JUnit), 65
méthode assertFalse (JUnit), 65
méthode assertNotNull (JUnit), 65
méthode assertNotSame (JUnit), 65
méthode assertNull (JUnit), 65
méthode assertSame (JUnit), 66
méthode assertTrue (JUnit), 66
méthode asyncExec, 162
méthode createPages (MultiPageEditor), 269
méthode createPartControl, 277
méthode destroy, 131
méthode dispose, 156
méthode doFinish (SampleNewWizard), 268, 269
méthode doGet, 202
méthode doPost, 202
méthode doSave, 270
méthode doSaveAs, 270
méthode equals, 65
méthode factorial, 76-78

méthode get
  description, 71
  objectif, 66, 67, 72
méthode getChecked, 186
méthode getDefault (Display), 162
méthode getEditor, 270
méthode getFirstElement, 277
méthode getIDsOfNames, 193
méthode getMessage, 37
méthode getSelection, 182
méthode getSelectionIndices, 165
méthode getText, 178
méthode getWriter, 202
méthode handleEvent (Listener), 178
méthode init, 260
méthode main
  création automatique, 15
  création de méthodes vides, 30
  création de méthodes vides (stubs), 32
méthode makeActions, 277
méthode openContentStream, 269
méthode openInformation
  (MessageDialog), 261
méthode pageChange, 271
méthode paint (Graphics), 134
Méthode paintComponent (JPanel), 137
méthode println, 199, 202
Méthode processEvent, 145, 146
méthode set
  description, 71
  objectif, 66, 67, 73
méthode setBounds, 161, 179
méthode setChecked, 186
méthode setImage, 186
méthode setIncrement, 184
méthode setLayout (shell), 161
méthode setLookAndFeel (UIManager),
  139, 139
méthode setMaximum, 184
méthode setMenuBar, 173
méthode setMessage, 37
méthode setMinimum, 184
méthode setPageIncrement, 184
méthode setPosition, 184
méthode setSelection, 183
méthode setText, 158
méthode setUp, 69, 72
méthode setVisible, 137
méthode show, 135
méthode sort, 271
méthode start, 131
méthode stop, 131
méthode syncExec, 162
méthode System.out.println, 20
méthode tearDown, 69
méthode widgetDefaultSelected
  (SelectionListener), 160
méthode widgetSelected
  (SelectionListener), 160
méthodes
  arguments et, 19
  assertion, 65
  assistant de code, 30
  changement de signature, 54
  création, 29-31, 145, 146
  déplacement, 52
  exceptions pour, 48
  Getter/Setter, 36, 37, 37
  méthodes déléguées, 36
  méthodes modifiées, 294, 296
  ouvrir une déclaration, 55
  paramètres pour, 48, 296
  points d'arrêt et, 85, 86
  recherche de, 57
  sauvegarde de fichiers et, 32
  surcharge, 33-35, 296, 299
  test à la volée, 21
méthodes d'allocation
  objectif, 66
méthodes déléguées, 36
méthodes getter, 36, 37, 296
méthodes modifiées, 294, 296
méthodes setter, 36, 37, 296
Microsoft Word, 9
mise en valeur syntaxique, 117
mises à jour, vérification, 289
mode d'insertion habile(éditeur du JDT),
  297
modèles
  création, 231-236
  édition, 59
  élement de type form beans et, 222
  getters/setters, 296
modèles de méthode
  création, 30
modification de code durant une session de
  débogage, 90-92
module additionnel V4ALL, 167-169
module V4ALL, 142-148
modules

GUI et, 142
modules (CVS), 94
modules logiques, 94
modules physiques, 94
modules virtuels, 94
modules, CVS et, 94, 98
mot réservé taskdef (Ant), 124
mot réservé typedef (Ant), 124
moteur de recherche, 192
Motif (look-and-feel), 139, 139, 141
Mozilla, 300
MultiPageEditor.java, 269, 272
MultiPageEditorContributor.java, 269

N

navigateurs
commande de navigation, 193
élément d'interface et, 193
Java et, 131
nettoyage, 221
nœuds, 185-186
noyau de la plateforme, 7
numéros de port, 198

O

Object Linking and Embedding (OLE), 9, 192, 193
Object Technologies International (OTI), 2, 3, 6
objet Display
méthode getDefault, 162
SWT et, 151, 152, 156
objet Event, 178, 181
objet Listener, 188
objet menuBar, 173
objet MenuItem, 172, 172
objet OleAutomation, 193
objet org.eclipse.editors.custom.StyledText, 269
objet org.eclipse.editors.ui.text.TextEditor, 269
objet SelectionListener, 173
objet Shell, 152, 187
objet SWT OleFrame, 193
objets
création, 38, 39
instanciation, 208,
objets IEditorPart, 270
OLE (Object Linking and Embedding), 9, 192, 193
OleControlSite (objet SWT), 193
OleFrame (objet SWT), 193
onjet SWT OleControlSite, 193
open source
contexte de test, 65
copyleft, 4
Eclipse et, 2, 4
option Chemin de compilation Java, 237
option Créer le fichier de compilation Ant, 120
option de menu Aide -> A propos de, 176
option de menu Aide -> A propos de Plateforme Eclipse, 246
option de menu Code Generation-> Générer le Code pour SWT, 167
option de menu Editer -> Annuler, 108
option de menu Equipe -> Mettre à jour, 102
option de menu Equipe -> Valider, 99, 104
option de menu Exécuter, 21
option de menu Exécuter -> Ajouter/supprimer un point d'arrêt, 76
option de menu Exécuter -> Ajouter/Supprimer un point d'arrêt dans une méthode, 86
option de menu Exécuter -> Déboguer, 91
option de menu Exécuter -> Déboguer en tant que -> Application Java, 77
option de menu Exécuter -> Exécuter, 46, 77, 252
option de menu Exécuter -> Exécuter en tant que -> Application Java, 26
option de menu Exécuter -> Outils externes -> Outils externes, 127
option de menu Exécuter en tant que -> Applet Java, 132
option de menu Exécuter en tant que -> Application Java
application AWT, 135
création de méthodes, 31
exécution du code, 19
option de menu Exécuter en tant que -> Test JUnit, 73
option de menu Exécuter en tant que ->Application Java
application V4ALL, 147
option de menu Fenêtre -> Afficher la vue -> Autre -> Ant, 127
option de menu Fenêtre -> Afficher les éditeurs, 9

option de menu Fenêtre -> Basculement vers un éditeur..., 9
option de menu Fenêtre -> Masquer les éditeurs, 9
option de menu Fenêtre -> Ouvrir la perspective, 10
option de menu Fenêtre -> Ouvrir la perspective -> Java, 10, 12, 12, 79
option de menu Fenêtre -> Personnaliser la perspective, 61
option de menu Fenêtre -> Préférences
configurer Ant, 124
option de menu Fenêtre -> Verrouiller les barres d'outils, 58
option de menu Fermer le projet, 27
option de menu Fichier -> Exporter, 280
option de menu Fichier -> Importer..., 26
option de menu Fichier -> Nouveau -> Classe, 15
option de menu Fichier -> Nouveau -> Page de testeur de code, 20
option de menu Fichier -> Nouveau -> Projet
création de plug-in, 254
nouveaux projets Tomcat, 211
projets de plug-in, 247
option de menu Fichier -> Nouveau -> Projet...
création de projets Java, 13
option de menu Fichier -> Sauvegarder, 273
option de menu Fichier -> Sauvegarder sous, 273
option de menu File (Fichier)-> Open External File (Ouvrir un fichier externe), 289
option de menu Generation de code-> Génération de code pour Swing, 146
option de menu Help (Aide) -> Software Updates (Mises à jour de logiciel) -> Manage Configuration (Gestion de la configuration), 288
option de menu Importer..., 26
option de menu Inspecter, 89
option de menu Navigate (Naviguer)-> Open Call Hierarchy (Ouvrir une hiérarchie d'appel), 293
option de menu Navigate (Naviguer)-> Open Declaration (Ouvrir une déclaration), 298
option de menu Naviguer -> Ouvrir le Javadoc externe, 48
option de menu New (Nouveau) -> Class (Classe)
création de classes, 285
option de menu New Menu -> New Action, 261
option de menu Nouveau -> Autre, 68, 237, 241
option de menu Nouveau -> Classe
classe englobante, 34
création de classes, 226
plusieurs classes et, 33
option de menu Nouveau -> Dossier, 206
option de menu Nouveau -> Dossier source, 224
option de menu Nouveau -> Emplacement du référentiel, 97, 100
option de menu Nouveau -> Fichier
création de documents, 203
fichier build.xml, 119
fichier de compilation Ant, 116
nouveaux projets JSP, 211
option de menu Nouveau -> Package, 225
option de menu Nouveau -> Projet, 265
option de menu Ouvrir la hiérarchie des types, 12, 19, 55
option de menu Ouvrir une déclaration, 55
option de menu Projet -> Propriétés, 223, 237
option de menu Projet -> Recompiler tout, 91, 91, 235
option de menu Projet -> Recompiler un projet, 235
compilation du code exécutable, 39
construction de servlets, 203
scrubbing et, 235, 235
option de menu Propager les modifications -> Déplacer..., 52
option de menu Propager les modifications -> Renommer..., 49, 50
option de menu Propriétés, 87
option de menu Rechercher -> Aide..., 56
option de menu Rechercher -> Fichier..., 56
option de menu Rechercher -> Java..., 56
option de menu Rechercher -> Rechercher..., 56
option de menu Réserver en tant que, 101
option de menu Réserver en tant que projet, 101, 112

option de menu Run (Exécuter)–> Run As (Exécuter en tant que)–> Java Application (Application Java), 286
option de menu Run (Exécuter)–> Run As (Exécuter en tant que) –> JUnit Plug-in Test (Unité de test de plug-in JUnit), 286
option de menu Search (Rechercher)–> File (Fichier), 289
option de menu Source –> Entourer d'un bloc try/catch, 36, 38
option Dépendances circulaires, 235
option Désélectionner le jeu de documents, 61
option Equipe –> Afficher dans l'historique des ressources, 103
option Equipe –> Ajouter au contrôle de version, 99
option Equipe –> Appliquer un correctif, 109
option Equipe –> Baliser en tant que version, 110
option Equipe –> Branche, 113
option Equipe –> Créer un correctif, 109
option Equipe –> Fusionner, 114
option Equipe –> Partager le projet, 97
option Equipe –> Synchroniser avec le référentiel, 105
option Exécuter –> Déboguer en tant que –> Plan de travail d'exécution, 253
option Exemple Menu –> Exemple d'action, 252
option Fenêtre –> Afficher la vue, 61
option Fenêtre –> Afficher la vue –> Autre, 278
option Fenêtre –> Afficher la vue –> Autre –> CVS –> Console CVS, 99
option Fenêtre –> Ouvrir la perspective –> Autre..., 96
option Fenêtre –> Ouvrir la perspective –> Navigation Java, 56
option Fenêtre –> Personnaliser la perspective, 209, 261
option Fenêtre –> Personnaliser la perspective –> Autre, 252
option Fenêtre –> Préférences –> Plan de travail –> Décorations d'intitulés, 99
option Fenêtre –> Réinitialiser la perspective, 58
option Inspecter, 81
option Java –> Débogage –> Filtrage de progression, 80
option Java –> Editeur –> Modèles, 61
option Java –> Génération de code –> Code et commentaires, 59
option Java –> Module de formatage de code, 59
option Naviguer –> Ouvrir un type de la hiérarchie, 55
option Nouveau –> actionSet, 256
option Ouvrir avec –> Editeur de texte, 203
option Projet –> Générer le Javadoc..., 48
option Refactor (Propagation des modifications) –> Change Method Signature (Changer la signature de la méthode), 296
option Refactor –> Introduce Factory, 294
option Replacer par –> Une autre branche ou version, 112
option Source (Source) –> Comment (Commentaire), 297
option Source (Source) –> Generate Getter and Setter (Générer les méthodes Getter et Setter), 296
option Source (Source)–> Add Block Comment (Ajouter un bloc de commentaire, 297
option Source (Source)–> Add Constructor from Superclass (Ajouter un constructeur de la superclasse), 296
option Source (Source)–> Generate Constructor Using Fields (Génération de constructeur à partir des données membres), 293
option Source (Source)–> Generate Delegate Methods (Générer les méthodes déléguées), 296
option Source (Source)–> Override/Implement Methods (Remplacer/Implémenter les méthodes), 296
option Source (Source)–> Remove Block Comment (Enlever un bloc de commentaire), 298
option Source –> Ajouter un constructeur de la superclasse, 36
option Source –> Générer des méthodes déléguées, 36
option Source –> Générer les méthodes Getter et Setter, 36, 37

option Source –> Mettre en commentaire, 36
option Source –> Remplacer/Implémenter les méthodes..., 34
option Source –> Supprimer la mise en commentaire, 36
option Team (Equipe)–> Show Annotation (Afficher les annotations), 300
option Window (Fenêtre)–> Customize Perspective (Personnaliser la perspective), 291
option Window (Fenêtre)–> Preferences (Préférences)–> Java –> Editor (Editeur)–> Typing (frappe), 296, 298
option Window (Fenêtre)–> Show View (Afficher la vue), 298
Oracle, 3
ordre (tri), 44
OTI (Object Technologies International), 2, 3, 6
outil de compilation Ant
  configuration, 124-127
  détection des erreurs dans les fichiers, 127-128
  fichier build.xml, 218
  Fichiers JAR et, 118-123
outil de construction Ant
  Eclipse 3.0 et, 299
outils de compilation (voir outil de compilation Ant)

P

page Contenu de plug-in simple, 254
page Fenêtre –> Préférences –> Java –> Compilateur –> Chemin de compilation, 235
page Fenêtre –> Préférences –> Java –> Génération de code –> Code et commentaires, 59
page Générateurs de code de plug-in, 248, 254, 265, 273
page Plug-in et fragments déployables, 280
page Sélection du point d'extension, 256
page Structure de project de plug-in, 265, 265
page Structure du projet de plug-in, 254
page Window (Fenêtre)–> Preferences (Préférences)–> Editors (Editeurs), 289
page Window (Fenêtre)–> Preferences (Préférences)–> Workbench (Plan de travail)–> Editors (Editeurs)–> Quick Diff, 290, 299
page Window (Fenêtre)–> Preferences (Préférences)–> Java –> Code Generation (Génération de code)–> Code and Comments (Code et commentaires), 296
page Window (Fenêtre)–> Preferences (Préférences) –> Install/Update (Installation/Mise à jour)–> Automatic Updates (Mises à jour automatiques), 289
pages de testeur de code, 20, 21
pages web, applets et, 132
paquetage java.awt, 131, 134
paquetage org.eclipse.swt.events, 158
paquetage org.eclipse.swt.layout, 158
paquetage org.eclipse.swt.ole.win32, 193
paquetages
  création, 38-39
  déploiement de plug-in, 280
  importation, 21
  pages de testeur de code et, 20
  recherche de, 57
  renommage, 51, 51
  selection, 225
  suggestions de Quick Fix, 23
  visualiser, 12
@param (Javadoc), 48
paramètres, 48, 296
PDE (Plug-in Development Environment)
  Eclipse et, 3
  Eclipse 3.0 et, 300
  éditeurs multipage et, 265-273
  généralités, 247-251
  plan de travail d'exécution, 251-253
perspective Débogage
  éditeurs, 79
  exécution de Tomcat et, 212
  généralités, 78
  inspection de valeurs, 81
  plan de travail et, 10
perspective Exploration du référentiel CVS, 96, 99
perspective Java
  ajout de plug-in dans, 262
  basculement à partir du débogueur, 79
  description, 12, 13
  partage de projet, 97
  selection, 10

vue Hiérarchie, 55
perspective Ressource, 11, 263
Perspective V4ALL, 144
perspectives
ajout de raccourcis, 96
basculement, 79
Eclipse 3.0, 283
généralités, 9-11
par défaut, 11
personnalisation, 61, 262-263
Perspective V4ALL, 144
restauration, 58
selection, 10
Photon QNX, 151, 157
placement en ligne (row layout), 158
plan de travail
fenêtre d'édition, 9
plan de travail d'exécution, 251-253, 261, 278
plans de travail (voir aussi plan de travail d'exécution)
Eclipse 3.0 et, 291
généralités, 7, 8
vues et, 9
plateforme Eclipse, 1-9
plateforme universelle (voir Eclipse)
plug-in (voir aussi plug-in PDE V4ALL)
actions et, 259-260
assistant pour, 247
CDT comme, 6
classes et, 247
configuration, 248
création, 253-256
création de vues, 273-279
déploiement, 280-281
Eclipse et, 3, 6
Eclipse 3.0 et, 292, 292
écriture de code pour, 260-261
éditeurs multipage, 265-273
espace de travail et, 8
fenêtre d'édition, 9
fichiers JAR et, 247
GUI et, 143
installation, 6
JDT comme, 6
jeux d'actions et, 256, 258
menus et, 258, 258
noyau et, 7
numéros de version et, 245
personnalisation de perspectives, 262-263
plan de travail d'exécution, 251-253
plug-in Easy Struts, 236-243
plug-in Tomcat Sysdeo, 209-216, 236
points d'extension et, 247
Quick Diff, 286
support d'équipe comme, 8
test avec JUnit, 300
plug-in CDT, 6
plug-in Easy Struts, 236-243
plug-in Tomcat Sysdeo, 209-216, 236
plug-in Xerces, 292
point (.), 19, 31
point d'extension org.eclipse.ui.actionSets, 255, 256
point d'extension org.eclipse.ui.perspectiveExtensions, 262, 262
point-virgule (;), 18, 296
points d'arrêt
compteur d'occurrences, 81-84, 86
configuration, 84-87
débogage de projets web, 216
Enlèvement, 76
exécutiondu code en pas à pas, 80
exécution jusqu'à, 81
gestion, 79
ignorer, 77
instauration, 76-79
points d'arrêt d'exception, 87-87
points d'arrêt de ligne de code, 85, 87
points d'arrêt de méthode, 86
points d'extension
construction au moyen de, 251
jeux d'actions et, 255, 256
plug-in et, 247, 262
points de contrôle, 85
préférences, exportation, 64
ProgressBar (contrôle SWT), 152
projecs
classpaths et, 43
fichier .project, 99
projets (voir aussi projets Java)
chemin de classe et, 13
chemin de compilation et, 39, 39
classes et, 32
compilateur du JDT, 299
CVS modules et, 94
débogage et, 215, 216

Eclipse et, 8
éditeur V4ALL, 143
exportation, 13
fermeture de, 27
fichier web.xml, 214
fichiers et, 268
fichiers JAR et, 43
fichiers manifestes de plug-in et, 247
gestion, 25-27
importation, 26
JSP, 211
navigation, 12
partage, 97-99
PDE et, 247, 247, 247
plug-in Easy Struts, 236
plusieurs paquetages dans, 39
recherche, 57, 299
référentiel CVS et, 96-114
réservation, 100, 101
restauration, 61
selection, 211
spécifications et, 284
vues et, 9, 9, 25
projets Java
création, 11-14, 118-123
Eclipse 3.0 et, 284-288
pages de testeur de code et, 20, 21
propagation des modifications, 48-54, 296
Propager les modifications –> Extraire une interface, 53
Propriété du texte (vue Propriétés), 144
Propriété Evénements (vue Propriétés), 146
propriété id (menus), 258
propriété label
actions, 259
jeux d'actions, 256
menus, 258
propriété menubarPath (actions), 259
propriété msg (JSP), 208, 209
propriété name
actions, 259
menu, 258
propriété toolbarPath (actions), 259
propriété toolTip (actions), 259
propriété visible (jeux d'actions), 256
propriétés
affecter les, 119
bean formulaires et, 238
outil de compilation Ant, 124, 125
propriétés de build.compiler, 119
protocole pserver, 95, 97
protocole SSH (secure shell), 95

Q

QNX Software Systems, 2
Quick Diff, 286, 290
Quick Fix, 22-25

R

raccourci Alt+F, 172
raccourci Ctrl+Alt+H, 293
raccourci Ctrl+T, 293
raccourci Pomme+F, 172
raccourcis
assistant au codage, 125
assistant de code, 19, 30, 48, 61
création, 61
déclarations, 298
esperluète et, 172
exécution du code en pas à pas, 80, 82
fonctions d'édition, 290
hiérarchie d'appel de méthode, 293
lancement d'Eclipse et, 5
perspectives et, 79, 96
Quick Diff, 290
validation de fichiers, 99
vue rapide Hiérarchie, 293
Rational Software3, 2
récepteur d'événements
menus et, 173
récepteurs
sélection, 186
récepteurs d'événements
item de barre d'outils et, 178
récepteurs d'événements (listeners)
boutons et, 160
listes et, 165
recherche, 56-57, 299
recompilation automatique, 288
Red Hat, 2
références
mise à jour automatique, 48
recherche de, 57
référentiel CVS
ajout d'un emplacement, 96, 97
balisage de version, 110-112
création de branches, 112
création de correctifs, 108, 110
mise à jour de code, 101-103
partage de projet, 97-99

réservation de projets, 100, 101
stockage de fichier, 94
synchronisation de code, 105-108
validation de fichiers, 99, 99
registre, 5, 246
registre des plug-in, 246
renommage d'éléments, 48-51
répertoire bin, 198
répertoire conf (Tomcat), 212, 214
répertoire CVSROOT, 99
répertoire de classes
dépendances de génération et, 235
exemple d'application Struts, 223
fichiers class et, 203
localisation du code compilé, 204
répertoire WEB-INF et, 200
répertoire de sortie
exemple de Struts, 223
nettoyage et, 221
répertoire de classes comme, 217
scrubbing et, 235
sélection par défaut, 204
répertoire de travail, 142
répertoire des modules, 142
répertoire des plug-in, 209
répertoire eclipse, 142
répertoire features, 142
répertoire icons, 245
répertoire lib
exemple d'application Struts, 223
fichier struts.jar, 224
fichiers JAR et, 245
répertoire WEB-INF et, 200
répertoire plugins, 245
répertoire src
assistant du PDE et, 267
création de vues et, 276
exemple d'application Struts, 226, 229
exemple de Struts, 224
fichier .java, 232
répertoire WEB-INF
création de servlets, 203, 206
exemple d'application Struts, 223
fichier web.xml dans, 217
fichiers tld, 224
Tomcat et, 200
répertoire webapps
déploiement d'applications pour le web, 218
exemple d'application Struts, 223
fichiers WAR, 216
serveur Tomcat et, 198, 201
répertoires (voir aussi répertoires contenant les fichiers compilés)
fichiers .class et, 39
code source, 224
liés, 205, 207, 209
serveur Tomcat web et, 200
serveur web Tomcat et, 197
répertoires liés, 205, 207, 209
requêtes GET (HTTP), 202
requêtes HTTP GET, 202
requêtes HTTP POST, 202
requêtes POST (HTTP), 202
ressources
éditeurs et, 9
espace de travail d'Eclipse et, 8
libération d'espace disque, 288
libération des, 156
perspectives et, 11
RowLayout (classe SWT), 163

S

SampleView.java, 276, 277
Sash (contrôle SWT), 152
Scale (contrôle SWT), 152
scriptlets, 199, 208
ScrollBar (contrôle SWT), 152
scrubbing, 235
sécurité, Eclipse 3.0 et, 292
sélecteur d'option, 180
Selection (événement SWT), 178
séparateur de barre d'outils, 177
séparateur de menu, 172
séparateurs de menu, 258
serveur d'aide (Eclipse), 9
serveur Tomcat web
numéro de port, 198
serveur d'applications Tomcat
Développement d'applications pour le Web et, 197
serveur web Tomcat
décodage de fichier, 206
plug-in Sysdeo, 209-216
structure de répertoire, 200
serveur web Tomcat web
fichier servlet.jar, 202
serveurs CVS, 95, 97, 99
serveurs web (voir serveur web Tomcat)
servlet action

création, 241
création de contrôleurs, 228-231
extension .do et, 228
fichier ActionResources.properties, 231
plug-in Easy Struts, 241
servlets
architecture MVC, 221
comme controleurs, 221
création, 201-204
création avec gestion des répertoires, 204-207
débogage, 215, 216
developpement de multiples, 44
fichiers WAR et, 217, 218
plug-in Tomcat Sysdeo et, 212-215
servlets action
extension .do et, 229
exemple d'application Struts, 234
plug-in Easy Struts, 241
répertoire de classe, 229
sessions SWT, 151
Shell (contrôle SWT), 152
shells
barre d'outils et, 176
exemple d'arbre, 185
SWT et, 152
Shift+Entrée, 290
Shift+F5 (avancer d'un pas avec filtres), 79
signatures de méthode, 54
signe plus (+), 109, 186
Slider (contrôle SWT), 152
smart insert mode, mode d'insertion habile (éditeur JDT), 286
Solaris Motif, 150, 151, 157
souris, clic, 279
sous-fenêtre Détails (vue Variables), 79
suffixe _SRC, 68
Standard Widget Toolkit (voir SWT)
style d'option de menu SWT.CASCADE, 172
style de barre d'outils SWT.SHADOW_OUT, 177
style de bouton SWT.ARROW, 159
style de bouton SWT.TOGGLE, 159
style de menu SWT.BAR, 171
style de menu SWT.LEFT_TO_RIGHT, 171
style de menu SWT.NO_RADIO_GROUP, 171
style de menu SWT.POP_UP, 171
style de menu SWT.RIGHT_TO_LEFT, 171
style SWT.APPLICATION_MODAL, 187
style SWT.BORDER
arbres, 184
barre d'outils, 176
boîtes de dialogue, 187
boutons, 158
contrôles texte, 159
curseurs, 180
étiquettes, 155
listes, 165
style SWT.CENTER
boutons, 159
contrôles texte, 159
étiquettes, 155
style SWT.CHECK
arbres, 184
boutons, 159
item d'outil, 177
option de menu, 172
style SWT.CLOSE, 188
style SWT.DIALOG_TRIM, 188
style SWT.DROP_DOWN
item d'outil, 177
menus, 171
style SWT.FLAT
barre d'outils, 176
boutons, 159
style SWT.HORIZONTAL
barre d'outils, 177
curseurs, 180
style SWT.H_SCROLL
arbres, 184
boîtes de dialogue, 188
listes, 165
style SWT.LEFT
boutons, 159
contrôles texte, 159
étiquettes, 155
style SWT.MAX, 188
style SWT.MIN, 188
style SWT.MULTI
arbres, 184
contrôles texte, 159
listes, 165
style SWT.NO_TRIM, 188
style SWT.PUSH
boutons, 159
option de menu, 172
style SWT.RADIO
boutons, 159

item d'outil, 177
option de menu, 172
style SWT.READ_ONLY de contrôle texte, 159
style SWT.RESIZE, 188
style SWT.RIGHT
barre d'outils, 176
boutons, 159
contrôles texte, 159
étiquettes, 155
style SWT.SEPARATOR
étiquettes, 155
item d'outil, 177
option de menu, 172
style SWT.SINGLE
arbres, 184
contrôles texte, 159
listes, 165
style SWT.TITLE, 188
style SWT.VERTICAL
barre d'outils, 177
curseurs, 180
style SWT.V_SCROLL
arbres, 184
listes, 165
style SWT.WRAP
barre d'outils, 176
contrôles texte, 159
étiquettes, 155
Sun Microsystems
serveur web Tomcat, 197
AWT et, 149
JDK, 300
look-and-feel, 150
look-and-feel Java, 139, 141
superclasses, 36
support d'équipe (Eclipse), 8, 8
surcharge de méthodes, 33-35, 296, 299
SuSE, 2
Swing
AWT et, 134
GUI et, 136-142
Java et, 149, 150
module V4ALL et, 143, 148
modules pour, 142
SWT et, 149
SWT (Standard Widget Toolkit)
arbres, 184-186
barre de menu et, 176-180
boîtes de dialogue, 187-192
boutons, 158-162
chemins de compilation, 151, 151
composites et gestionnaires de placement, 163-164
configurations d'exécution, 157
contrôles et options, 152-155
curseurs, 180-184
Eclipse 3.0 et, 292, 300
Java et, 150
libération des ressources, 156
listes, 165-167
look-and-feel, 138
menus, 171-176
objectifs, 149
ouverture de IE, 192-195
plan de travail et, 7, 7
styles d'étiquette, 155
V4ALL et, 167-169
widgets et threads, 162
swt-pi.jar file, 151
SWT.NONE (constante), 163
SWT.PUSH style
item d'outil, 177
Sybase, 3
symbole arrobas (@), 48
symbole dollar ($), 61
synchronisation
code, 105-108
fichiers, 98
système d'exploitation
gestionnaire des fenêtres, 152
Swing, 138
système d'exploitation HP-UX, 7
système d'exploitation Mac OS X
configuration d'exécution, 157
OLE et, 192
raccourci Pomme+F, 172
SWT et, 150, 151
système d'exploitation OS X pour Mac
portage d'Eclipse, 7
système d'exploitation Solaris, 7
systèmes d'exploitation
AWT et, 149
initialisation des variables d'environnement, 198
plan de travail et, 7
support CVS, 93
SWT et, 7, 150

T

TabFolder (contrôle SWT), 152
TabItem (contrôle SWT), 152
Table (contrôle SWT), 152
TableColumn (contrôle SWT), 152
TableItem (contrôle SWT), 152
tcsh shell (Unix), 198
test
  applets, 132
  développement de plug-in et, 300
  méthodes à la volée, 21
  plan de travail d'exécution, 251-253, 261
test de bon fonctionnement
  conformité, 75
  création d'une application de test, 67
  création d'une application test, 66
  JUnit et, 65, 66, 68-75
test de conformité, 75
testing
  contexte open source, 65
Text (contrôle SWT), 152
texte d'une étiquette
  styles SWT, 155
threads, widgets et, 162
thumb (glissière), 183
fichiers .tld (Tag Library Definition), 224, 236
TogetherSoft3, 2
Tomcat
  installation et configuration, 197-199
ToolBar (contrôle SWT), 152
ToolItem (contrôle SWT), 152
Tracker (contrôle SWT), 152
traitement des erreurs, 127-128
Tree (classe SWT), 184
Tree (contrôle SWT), 152
TreeItem (contrôle SWT), 152
tri (classement), 271
tri (ordre), 44

U

UI thread, 162

V

VA4J (Visual Age for Java), 3
validation
  code, 102, 103, 105
  fichiers, 99, 99
variable ANT_HOME, 124
variable d'environnement
  CATALINA_HOME, 198, 210
variable d'environnement JAVA_HOME, 198
variable d'environnement PATH, 198
variables (voir aussi variables locales)
  débogage, 10, 79, 81
  édition de valeurs en cours de débogage, 89
  suffixe de code source, 68
  variables de classpath, 44, 67, 68
variables d'environnement
  CATALINA_HOME, 198, 210
  Eclipse 3.0 et, 299
  JAVA_HOME, 198
  PATH, 198
variables de classpath, 44, 67, 68
variables locales
  contexte d'exécution empilés et, 75
  conversion de membres de classe, 54
  débogage, 78, 82
  éditeur du JDT et, 298
  points de contrôle et, 86
verrouillage optimiste, 94
verrouillage pessimiste, 94
version diffusable, 93
versions
  balisage, 110-112
  branches et, 112
  correctifs et, 110
  CVS et, 94
  Eclipse, 2, 5
  Eclipse 3.0, 283
  outil de compilation Ant, 124
  plug-in et, 245
  règles de balisage, 111
  support d'équipe et, 8
  versions beta, 4
  versions diffusables, 4, 5
versions bêta
  contrôle de code source et, 93
  Eclipse, 4
  Eclipse 3.0, 283
versions diffusables, 4
versions en phase d'intégration, 4
versions expérimentales, 5
versions stables, 4
Visual Age for Java (VA4J), 3
vue Affichage, 78, 79

vue Annotation CVS (Annotation CVS), 300
vue Ant, 127
vue Comparaison du source Java, 106
vue Console
  console CVS et, 99
  Eclipse 3.0 et, 288
  exemple de méthode factorial, 76
  pages de testeur de code, 21
  perspective Débogage, 79
  perspective Java, 13
  Quick Diff, 286
  vue Synchronisation et, 98
vue CVS Annotation (Annotation CVS), 300
vue Débogage
  boutons dans, 78, 81, 212
  compteur d'occurrences de point d'arrêt, 82
  exécution du code en pas à pas, 79
  généralités, 78
vue Declaration (Déclaration), 298
vue Error log (Liste des erreurs), 298
vue Expressions
  débogage et, 89
  fonctionnalité, 79
  inspection de valeurs, 81, 82
  perspective Débogage, 78
vue Hiérarchie
  accès aux membres d'une classe, 19
  depicted, 55
  description, 12
  Eclipse 3.0 et, 293
vue Historique des ressources CVS, 103
vue Javadoc, 298
vue JUnit, 73
vue Membres, 56
vue Navigateur, 9, 25
vue Navigator (Navigateur), 284
vue Outline (Structure)
  Eclipse 3.0 et, 286
Vue Packages, 56
  assistant de code et, 19
  assistant JUnit, 68
  branches et, 114
  chemin de compilation et, 39
  classes et, 15, 34, 50, 213
  code et, 19, 20, 102
  configuration d'exécution, 156
  description, 12
  éditeurs et, 265, 273
  fichier build.xml et, 119
  fichier servlet.jar, 203
  fichier swt.jar, 150
  fichier web.xml, 206
  fichiers et, 99
  fichiers JAR et, 42
  jeux de documents, 61
  objectif, 10
  paquetages et, 38, 51
  plan de travail d'exécution, 252
  plugin.xml, 248
  projets et, 13, 25-26, 39, 61, 121
  projets Tomcat, 211
  recherche et, 57
  répertoire bin et, 40
  test de bon fonctionnement et, 73
  test et, 132
  V4ALL et, 144, 147
  versions et, 112
vue Points d'arrêt
  création d'un compteur d'occurrences, 82
  points d'arrêt sur une exception et, 87
  points de contrôle dans, 86
vue Points d'arrêts
  functionality, 79
  sous forme de vues empilées, 78
vue Problems (Problèmes), 284
vue Progress (Progression), 288
vue Projets, 56
vue Propriétés
  création d'actions, 259, 259
  feuille à dessin et, 144
  jeux d'actions et, 256
  menus, 258
  Propriété du texte, 144
  Propriétés Evénements, 146
vue Recherche, 57
vue Référentiels CVS, 100, 101, 111, 113
vue Structure
vue Structure
  description, 13
  description, 55
  fichier build.xml, 117, 121
  perspective Débogage, 79
vue Synchronisation, 98, 106, 106
vue Tâches, 13
vue Tasks (Tâches), 284
vue Variables, 78, 79, 89

vues
architecture MVC et, 221
création, 273-279
Eclipse 3.0, 283
Eclipse 3.0 et, 298
enlever des projets de, 25
généralités, 9-11
ré-ouverture, 9
spécification, 61
Struts et, 225-227

W

Webgain2, 2
widgets
contrôles natifs et, 150
définis, 152
Eclipse 3.0 et, 300
explorateurs et, 300
navigateur et, 193
threads et, 162
windows
fenêtres dans, 9

X

XML
bien écrit, 127
code xml du plug-in, 254
Eclipse 3.0 et, 292
éléments servlet, 203, 206, 214
erreurs de syntaxe et, 128
Fichier de compilation Ant, 116, 117
serveur d'aide et, 9
XML Buddy, 200, 203

Composé en FRANCE par JOUVE,
11, bd de Sébastopol, 75001 Paris - France
N° 358965B - Dépôt légal : Novembre 2004

	Débutant	Expérimenté	Spécialiste
Se former			
Agir vite			
Aide-mémoire			

O'REILLY®

Eclipse

Eclipse est un outil extraordinaire qui vient satisfaire les besoins exprimés de longue date par les développeurs Java : réunir dans une seule application un plan de travail, des outils de gestion de projet en équipe et des fonctionnalités évoluées de débogage. En outre, Eclipse peut être dopé au moyen de nombreux plug-ins disponibles sur l'Internet (Ant, Tomcat, etc.). Il offre donc une plateforme de développement particulièrement riche, extensible et ergonomique. Disposant d'une palette d'outils très étendue, le programmeur peut se concentrer exclusivement sur l'écriture de son code. Ceux qui n'ont jamais utilisé Eclipse auparavant verront leur productivité bondir d'un pas de géant.

Alors que la plupart des ouvrages consacrés à Eclipse traite uniquement du développement d'applications Java, cet ouvrage envisage Eclipse de façon globale et ne se limite pas à cet aspect. Il s'adresse à tous les programmeurs Java, du développeur désireux de maîtriser son EDI au programmeur qui n'en peut plus de devoir écrire son code dans Notepad ou vi.

On trouvera dans cet ouvrage :

- une présentation du JDT (Java Development Tools) et de l'espace de travail d'Eclipse ;
- l'intégration avec Ant, notamment la production de fichiers JAR ;
- le développement en équipe avec CVS ;
- les outils et les techniques de débogage à la disposition du programmeur ;
- le développement d'applications Swing ;
- l'utilisation de SWT (Standard Widget Toolkit) et la création de widgets ;
- le développement de servlets et de JSP ;
- l'intégration du serveur d'applications Tomcat ;
- le développement d'applications Struts ;
- le déploiement et le développement d'applications dynamiques et d'EJB ;
- le développement de plug-ins à l'aide du PDE.

Chaque sujet est abordé avec clarté et concision. Écrit par un programmeur pour des programmeurs, cet ouvrage aborde l'essentiel et seulement l'essentiel. Au début de chaque chapitre, l'auteur dresse un état des lieux des concepts théoriques sous-jacents avant de passer à la pratique en illustrant ses propos à l'aide d'exemples de code.

ISBN 2-84177-264-0

9 782841 772643 40 euros

www.oreilly.fr